Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 99 (2004) 49–86

www.elsevier.com/locate/entcs

Program Transformations under Dynamic Security Policies

Massimo Bartoletti[1] Pierpaolo Degano[1] Gian Luigi Ferrari[2]

Dipartimento di Informatica, Università di Pisa, Italy

Abstract

A new static analysis is proposed for programming languages with access control based on stack inspection. This analysis allows for various security-aware program optimizations. A novel feature of our static analysis is that it is parametric with respect to the security policy in force, so it needs not to be recomputed when the access rights are dynamically updated.

Keywords: Language-based security, Access control, Static analysis, Compiler optimizations

1 Introduction

Programming applications over wide area networks has emphasized issues that had received less priority when working over local area networks. One of these issues concerns security. In a wide area network no central authority can define and enforce policies which regulate accesses to resources. Applications are designed to be executed and interoperate with potentially malicious and untrusted components, e.g. components originated from different, possibly unknown, administration domains. Typically, components are developed and maintained by different providers and may be downloaded and linked together "on demand". At run-time, systems interleave computational activities

[1] Partially supported by EU project DEGAS and MIUR project MEFISTO.
[2] Partially supported by FET project PROFUNDIS.

doi:10.1016/j.entcs.2004.02.003

with meta-programming activities, such as dynamic linking, assembling and customization of libraries, that permit to reconfigure the application without having to restart it.

Security-aware programming languages, such as Java and C♯, have introduced programmable authorization-based models to determine when a principal can access a resource. These languages take access control decisions by inspecting the run-time call stack. A permission is granted, provided that it belongs to *all* the principals on the call stack. The so-called *privileged operations* are an exception: they are allowed to execute any code granted to their principal, regardless of the calling sequence. This access control mechanism is known as *stack inspection*.

Traditionally, stack inspection has been implemented with the *lazy evaluation* strategy: the call stack is only retrieved and inspected when access control tests are performed. This strategy has some drawbacks. First, the run-time overhead due to the analysis of stack frames may grow very high. Second, stack inspection deeply affects those interprocedural program transformations (i.e. method inlining) that may alter the structure of the call stack.

Another evaluation strategy for stack inspection is the *eager* one: the access control context is updated at each method call (and return). However, since security checks are statistically less frequent than cross-domain calls, production implementations prefer to adopt the lazy strategy.

A large amount of papers [2,3,4,6,9,10,12,17,21,28] witnesses the interest towards formally understanding and optimizing stack inspection. All these approaches share a basic assumption: the binding between code and permissions is made at class-loading time, and it cannot be modified at run-time.

However, starting from version 1.4.1, the Java security architecture allows for *dynamic security policies*: the binding between a class and its permissions *can* be deferred until the class is involved in an access control test. Still, the static binding of permissions is allowed.

In this paper, we introduce a new static analysis for stack inspection, improving over previous analyses of ours, see e.g. [3]. Our present proposal is specifically designed to tackle the issues raised by dynamic security policies. We represent programs by *control flow graphs*, an idealized model not tied to any particular language. These graphs are extracted from actual bytecode through available control flow analyses; they feature primitive constructs for method invocation, exceptions, and access control based on stack inspection.

Our analysis takes as input a control flow graph, and computes an abstract graph, whose nodes are pairs $\langle n, \gamma \rangle$ and edges are labelled by φ, where:

- n is a method invocation or return, or an access control test;

- γ is an *access control context*, i.e. the set of protection domains visited after the last privileged call (if any);
- φ is a traversability condition for the edge.

The traversability conditions are used to associate each abstract node $\langle n, \gamma \rangle$ with a predicate $\Phi(n, \gamma)$, telling which conditions the security policy must satisfy in order for the node to be reachable.

We prove our static analysis correct with respect to the operational semantics of control flow graphs: if the actual security policy allows for an execution leading to node n with access control context γ, then there exists a node $\langle n, \gamma \rangle$ in the abstract graph, and the security policy satisfies the condition $\Phi(n, \gamma)$.

A significant novelty of our approach is that the static analysis is *parametric* with respect to the security policy in force, hence it needs not to be recomputed each time the access rights are modified.

Our static analysis provides a formal support for some security-aware optimizations. In particular, we can detect (and remove):

- the redundant checks in a program, i.e. the checks which always pass;
- the dead code, which cannot be reached due to security restrictions;
- the inlineable method calls, i.e. those calls which can be safely replaced by a copy of the called method.

Indeed, method inlining can lead to dramatic performance improvements, for the following reasons. First, it reduces the overhead of dynamic dispatching; second, it allows the just-in-time optimizers to work on larger blocks of code, so making standard intraprocedural optimizations more effective.

Our static analysis establishes conditions on the security policy which ensure when these optimizations are valid.

2 Program model

We model programs as *control flow graphs* (CFGs for short) whose nodes represent access control tests, method invocations and returns, and whose edges represent the flow of control. We also consider a basic exception handling mechanism, with only one type of exceptions, no nested `try` blocks and no `finally` clauses. We do not define how CFGs are extracted from actual programs: this construction is well understood and algorithms and tools exist for it; see for example [14,19,26,27]. A full treatment of exceptions requires a tailored construction of the CFG, e.g. by the techniques presented in [8,23].

CFGs hide any data flow information, and are therefore approximated; typically, the conditional construct is rendered as non-deterministic choice.

This approximation is *safe*, in the sense that any actual execution flow is represented by a path in the CFG. However, the converse may not be true: some paths may exist which do not correspond to any actual execution. For instance, both branches of an "if" statement are represented, even in the cases when the same branch is always taken at run-time.

Dynamic dispatching in object-oriented languages is another source of approximation. When a program invokes a method on an object O, the run-time environment has to choose among the various implementations of that method. The decision is based on the actual class O belongs to, which is unpredictable at static time. To be safe, CFGs over-approximate the set of methods that can be invoked at each program point.

2.1 Syntax

Let $\mathcal{D}$ be a finite set of protection domains, and $\mathcal{P}$ be a finite set of permissions.

Definition 2.1 *A CFG $\langle N, E, Dom, Priv\rangle$ is an oriented graph, where:*

- *N is the set of nodes. Each $n \in N$ is associated with a label $\ell(n)$, describing the control flow primitive represented by the node. Labels give rise to three kinds of nodes:* `call` *nodes, representing method invocation,* `return` *nodes, which represent return from a method, and* `check`(P) *nodes, testing whether the permission P is granted by the security policy or not. A distinguished element $n_\varepsilon \in N$ plays the technical role of a single, isolated entry point.*
- *$E \subseteq N \times (N \setminus n_\varepsilon)$ is the set of edges. There are three kinds of edges:* call edges $n \longrightarrow n'$*, which model* interprocedural *flow,* transfer edges $n \dashrightarrow n'$*, which correspond to sequencing, and* catch edges $n \dashrightarrow_t n'$*, which correspond to exception handling. The last two kinds represent* intraprocedural *flow.*
- *$Dom : N \to \mathcal{D}$ is a mapping from nodes to protection domains.*
- *$Priv : N \to Bool$ tells whether a node enables its privileges or not.*

Each CFG is associated with a *security policy* $Perm : \mathcal{D} \to 2^{\mathcal{P}}$, which grants a set of permissions to each protection domain. The following treatment does not require security policies to be fixed over time. Hereafter, we will always abbreviate $Perm(Dom(n))$ with $Perm(n)$.

Definition 2.2 *The* methods *of a control flow graph $\langle N, E\rangle$ are the connected components of the undirected graph $\langle N, E'\rangle$, where E' is the symmetric closure of the set of intraprocedural edges in E. We call $\mu(n)$ the method to which node n belongs. The* entry points *of $\mu(n)$ are defined as:*

$$\varepsilon(\mu(n)) \;=\; \{\, m \in \mu(n) \mid \exists n' \in N.\; n' \longrightarrow m \,\}$$

2.2 Semantics

The operational semantics of CFGs is defined by a transition system, whose configurations are call stacks. Moreover, each state has a boolean tag which tells whether an exception is active (i.e. thrown and not caught yet). If no exception is active, a state is represented as sequence of nodes enclosed in square brackets: for example, $\sigma = [n_0, \ldots, n_k]$ is a state whose top node is n_k. If an exception is active, we append the symbol $\lightning$ to the sequence of nodes, i.e. $\sigma\lightning$ abbreviates $\langle \sigma, \mathit{true} \rangle$. Pushing a node n on a stack σ is written as $\sigma : n$.

The access control context of a state is defined as the set of protection domains visited after the last privileged call (if any). Notice that, in what follows, we will write x for a singleton set $\{x\}$, when unambiguous.

Definition 2.3 *We define the* access control context $\Gamma(\sigma)$ *of a state* σ *as:*

$$\Gamma([]) = \emptyset \qquad\qquad \Gamma(\sigma : n) = \Gamma(\sigma) \uparrow n$$

where, for each context γ *and* $n \in N$*, we define:*

$$\gamma \uparrow n \;\; = \;\; \begin{cases} \mathit{Dom}(n) & \mathit{if\ Priv}(n) \\ \gamma \cup \mathit{Dom}(n) & \mathit{otherwise} \end{cases}$$

Stack inspection is modeled by the minimal relation induced by the inference rules for the predicate $\vdash$ below. The set of permissions granted to a state is just the intersection of the permissions associated to its access control context. We prefer to formalize stack inspection using a double indirection with access control contexts and permissions, because our static analysis will be independent from the security policy in force.

Definition 2.4 *We say that a permission* P *is* granted *to a state* σ *under the security policy* Perm *iff* $\Gamma(\sigma) \vdash_{\mathit{Perm}} P$*, where:*

$$\frac{P \in \mathit{Perm}(D)}{D \vdash_{\mathit{Perm}} P} \qquad\qquad \frac{\gamma \vdash_{\mathit{Perm}} P \quad \gamma' \vdash_{\mathit{Perm}} P}{\gamma \cup \gamma' \vdash_{\mathit{Perm}} P}$$

The transition relation $\rhd$ between states is defined in Table 1. A *trace* on $\langle G, \mathit{Perm} \rangle$ is a derivation $\langle \sigma_0, \chi_0 \rangle \rhd \cdots \rhd \langle \sigma_k, \chi_k \rangle$. An *entry trace* is a trace where $\sigma_0 = [n_\varepsilon]$ and $\chi_0 = \mathit{false}$. The reachability relation $\rhd$ states when there is an entry trace on $\langle G, \mathit{Perm} \rangle$ which can lead to a given state:

$$\frac{}{\langle G, \mathit{Perm} \rangle \rhd [n_\varepsilon]} \qquad\qquad \frac{\langle G, \mathit{Perm} \rangle \rhd \sigma \quad \sigma \rhd \sigma'}{\langle G, \mathit{Perm} \rangle \rhd \sigma'}$$

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n'}{\sigma : n \rhd \sigma : n : n'} \ [\rhd_{call}] \qquad \frac{\ell(m) = \texttt{return} \quad n \dashrightarrow n'}{\sigma : n : m \rhd \sigma : n'} \ [\rhd_{return}]$$

$$\frac{\ell(n) = \texttt{check}(P) \quad \Gamma(\sigma : n) \vdash_{Perm} P \quad n \dashrightarrow n'}{\sigma : n \rhd \sigma : n'} \ [\rhd_{pass}] \qquad \frac{n \dashrightarrow_{\lightning} n'}{\sigma : n\lightning \rhd \sigma : n'} \ [\rhd_{catch}]$$

$$\frac{\ell(n) = \texttt{check}(P) \quad \Gamma(\sigma : n) \nvdash_{Perm} P}{\sigma : n \rhd \sigma : n\lightning} \ [\rhd_{fail}] \qquad \frac{n \not\dashrightarrow_{\lightning}}{\sigma : n\lightning \rhd \sigma\lightning} \ [\rhd_{propagate}]$$

Table 1
Operational semantics of CFGs.

2.3 Well-formed CFGs

We require our CFGs to obey some mild well-formedness constraints. We say that a CFG is *well-formed* iff, for each $n, n' \in N \setminus n_\varepsilon$:

$$\ell(n) = \texttt{check}(P) \implies \nexists n' \in N.\ n \longrightarrow n' \tag{1a}$$
$$\ell(n) = \texttt{return} \implies \nexists n' \in N.\ \langle n, n' \rangle \in E \tag{1b}$$
$$|\, \varepsilon(\mu(n)) \,| \ = \ 1 \tag{1c}$$
$$\mu(n) = \mu(n') \implies \mathit{Dom}(n) = \mathit{Dom}(n') \tag{1d}$$
$$\mathit{Priv}(n) \implies \ell(n) = \texttt{call} \tag{1e}$$
$$n \dashrightarrow n' \ \vee \ n \dashrightarrow_{\lightning} n' \implies n' \neq \varepsilon(\mu(n)) \tag{1f}$$
$$\mu(n_\varepsilon) = n_\varepsilon \tag{1g}$$

Constraints (1a) and (1b) ensure, respectively, that check nodes have no outgoing call edges and that return nodes do not have outgoing edges at all. Constraint (1c) states that each method has exactly one entry point. After this constraint, we abbreviate $n \longrightarrow \varepsilon(\mu(m))$ with $n \longrightarrow \mu(m)$. Constraint (1d) says that nodes in the same method are in the same protection domain. Note that constraints (1a)–(1d) reflect some peculiarities of Java-like bytecode: hence, CFGs extracted from bytecode always satisfy them. The other constraints are only technical, and help keeping the definition of our analysis short: (1e) says that only call nodes can be privileged; (1f) that intraprocedural and interprocedural edges do not overlap; (1g) states that the entry point n_ε has no outgoing intraprocedural flow (therefore, it only makes sense to have n_ε as a call node).

2.4 Adequacy of the model

We briefly discuss some of the differences between our model and the Java security model [13]. Similar considerations hold for the .NET model [22].

- Java allows for the dynamic instantiation of permissions (e.g. an application that asks the user for a file name and then tries to open that file). Such *parametric permissions* are of the form $P(x)$, where x ranges over the set of possible targets for the permissions of class P.
- in Java, a new thread upon creation inherits the access control context of its parent. When stack inspection is performed, both the context of the current thread and the contexts of all its ancestors are examined. In this way, a child thread cannot obtain an access which is not granted to its ancestors.
- our model only allows for *code-centric* security policies: permissions are granted to code according to its source, regardless of who is running it. JAAS [18], extends the Java security model by enabling *user-centric* access control policies, based on the principal who actually runs the code. Permissions can be granted to principals, and the `doAs` method allows a piece of code to be executed on behalf of a given subject. This is done by associating the (authenticated) subject running the code with the current access control context. Stack inspection ensures that subjects are taken into account when access control is performed (see e.g. [15] for a formal specification).
- we do not model some advanced features like reflection and native methods. Also, we do not consider the side effects of some "dangerous" permissions (e.g. `AllPermission`, which may even breach the whole security system by replacing the JVM system binaries). Besides deeply affecting security, these features reduce the effectiveness of any analysis which aims at determining statically the permissions granted to running code.

3 The Security Context Analysis

Given a CFG $G = \langle N, E, Dom, Priv \rangle$, our static analysis computes an abstract graph $G^\sharp = \langle N^\sharp, E^\sharp \rangle$ taking into account the evolution of the security contexts in the traces of G. An abstract node $n^\sharp \in N^\sharp$ is a triple $\langle n, \gamma, \chi \rangle$ (abbreviated as $n\gamma\chi$ when unambiguous). Intuitively, it represents a call stack with top node n, access control context γ and exception flag χ. An abstract edge $n^\sharp \xrightarrow{\varphi} m^\sharp$ models an execution that can flow from $n^\sharp$ to $m^\sharp$ if the security policy in force satisfies the condition φ. The root of $G^\sharp$ is $n^\sharp_\varepsilon = \langle n_\varepsilon, Dom(n_\varepsilon), false \rangle$.

Our analysis is specified by the inference rules in Table 2. The abstract graph is constructed starting from the root, and then applying the inference rules to obtain new abstract edges and nodes. Technically, this is a forward,

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n' \quad n\gamma \in N^\sharp}{n\gamma \rightarrow n'(\gamma \uparrow n \cup Dom(n'))}\ [\sharp_{call}] \qquad \frac{n\gamma\lightning \in N^\sharp \quad n \dashrightarrow_\lightning m}{n\gamma\lightning \rightarrow m\gamma}\ [\sharp_{catch}]$$

$$\frac{\ell(n) = \texttt{check}(P) \quad n\gamma \in N^\sharp \quad n \dashrightarrow m}{n\gamma \xrightarrow{\gamma \vdash P} m\gamma}\ [\sharp_{pass}] \qquad \frac{\ell(n) = \texttt{check}(P) \quad n\gamma \in N^\sharp}{n\gamma \xrightarrow{\gamma \nvdash P} n\gamma\lightning}\ [\sharp_{fail}]$$

$$\frac{\ell(n') = \texttt{return} \quad n \dashrightarrow m \quad n\gamma \rightleftarrows_\varphi n'\gamma'}{n\gamma \xrightarrow{\varphi} m\gamma}\ [\sharp_{return}] \qquad \frac{n\gamma \rightleftarrows_\varphi n'\gamma'\lightning \quad n' \not\dashrightarrow_\lightning}{n\gamma \xrightarrow{\varphi} n\gamma\lightning}\ [\sharp_{propagate}]$$

Table 2
Abstract semantics of control flow graphs.

monotone control flow analysis. The nodes are in $N \times 2^{\mathcal{D}} \times Bool$, and the edges are in $(N \times 2^{\mathcal{D}} \times Bool)^2 \times 2^{(2^{\mathcal{D}} \times \mathcal{P})}$. Since N, $\mathcal{D}$ and $\mathcal{P}$ are finite, then the abstract graph is finite, too.

Before explaining the rules in Table 2, it is convenient to introduce some terminology and notation. First, we define the notion of *paths* over abstract CFGs (definition 3.1). Definition 3.3 is used to determine which abstract returns (and exception propagations) match abstract calls. This allows us to single out paths that model "valid" abstract executions. Definition 3.2 introduces the concepts of *traversability* and *reachability*. Intuitively, a path is traversable under a security policy if the policy satisfies all the conditions on the path edges. An abstract node is reachable under the policy if there exists a path leading to that node which is traversable under the policy.

Definition 3.1 *A* path *from* $n_0^\sharp$ *to* $n_k^\sharp$ *in* $G^\sharp$ *is a sequence* $n_0^\sharp \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_k} n_k^\sharp$ *where, for each* $i \in 1..k$, $n_{i-1}^\sharp \xrightarrow{\varphi_i} n_i^\sharp \in E^\sharp$. *We denote with* $\Pi(n^\sharp, m^\sharp)$ *the set of all paths from* $n^\sharp$ *to* $m^\sharp$, *and with* $\Pi(n^\sharp)$ *the set of all paths from* $n_\varepsilon^\sharp$ *to* $n^\sharp$. *We write* $\Pi(G^\sharp)$ *to denote all the paths in the abstract graph* $G^\sharp$.

Definition 3.2 *Let* $n^\sharp \in N^\sharp$ *and* $\pi = n_0^\sharp \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_k} n_k^\sharp$. *We define:*

$$\Phi(n^\sharp) \;=\; \bigvee_{\pi \in \Pi(n^\sharp)} \Phi(\pi) \qquad\qquad \Phi(\pi) \;=\; \bigwedge_{i=1..k} \varphi_i$$

By convention, $\Phi(n^\sharp) = false$ *if* $n \notin N^\sharp$, *and* $\Phi(\pi) = true$ *if* $\pi = []$.

Definition 3.3 *Let* $n\gamma, n'\gamma'\chi' \in N^\sharp$. *We write* $n\gamma \rightleftarrows_\varphi n'\gamma'\chi'$ *when* $n \longrightarrow \mu(n')$, $\gamma' = (\gamma \uparrow n) \cup Dom(n')$, *and there exists a path from* $\varepsilon(\mu(n'))\gamma'$ *to* $n'\gamma'\chi'$ *s.t.*

$w(\pi) = 0$ and $\Phi(\pi) = \varphi$. The weight of paths and edges in $G^\sharp$ are defined as:

$$w(\pi) = \sum_{i=1}^{|\pi|} w(\pi[i-1] \to \pi[i]) \qquad w(n\gamma\chi \to n'\gamma'\chi') = \begin{cases} 1 & \text{if } n \longrightarrow n' \\ 0 & \text{otherwise} \end{cases}$$

Constraint (1f) guarantees that abstract edges representing interprocedural flow have unit weight, while intraprocedural ones have null weight. This constraint could be removed by labelling the abstract edges with their weight.

We now introduce the "concrete" counterpart of definition 3.2 above. The traversability condition on a trace is the conjunction of the (unevaluated) access control tests encountered. We formally relate the traversability of paths to that of traces in Theorem 3.6.

Definition 3.4 *Given a trace $\tau = \sigma_0\chi_0 \rhd_{\ell_1} \cdots \rhd_{\ell_k} \sigma_k\chi_k$, we define:*

$$\Phi(\tau) = \textstyle\bigwedge_{i=1..k} \Phi(\sigma_{i-1}\chi_{i-1} \rhd_{\ell_i} \sigma_i\chi_i)$$

$$\Phi(\sigma : n, \chi \rhd_\ell \sigma'\chi') = \begin{cases} \Gamma(\sigma : n) \vdash P & \text{if } \ell(n) = \texttt{check}(P),\ \ell = \mathit{pass} \\ \Gamma(\sigma : n) \nvdash P & \text{if } \ell(n) = \texttt{check}(P),\ \ell = \mathit{fail} \\ \mathit{true} & \text{otherwise} \end{cases}$$

We here provide some intuition about the rules in Table 2. Consider $\sharp_{call}$ first. If there is a call from node n to n', and the access control context before the call is γ, then the context after the call comprises the protection domain of n', plus either that of n if n is privileged, or γ if not. The rule $\sharp_{pass}$ says that, if a node n tests for permission P in the context γ, there is an edge labelled $\gamma \vdash P$ leading to m and preserving the context. Similarly, the rule $\sharp_{fail}$ says that there is an edge $\gamma \nvdash P$ leading to an exception. The rule $\sharp_{return}$ states that, if n' is a return and there is a call node n with context γ that *matches* the return node (see definition 3.3 below), then there is an edge from n to m while preserving the context γ. The rule $\sharp_{propagate}$ states that, if the call n with context γ can match a node n' while throwing an exception, then the exception is propagated to n.

Practical considerations suggest us that the size of an abstract graph does not grow exponentially, but it is actually linear in the number of nodes of the original CFG. In particular:

- the exponential factor $2^{\mathcal{D}}$ above only occurs when the number of protection domains is proportional to the number of nodes. Actually, the number of protection domains can be considered as a constant, because it depends on static properties of the loaded code (i.e. code origin and digital signatures).

- each security check gives rise to a bifurcation in the abstract graph. Then, our approximation to the number of abstract nodes hides an exponential factor in the number of checks. However, the number of security checks in CFGs is usually small: indeed, access control tests are only inserted to guard methods accessing critical resources.
- when some protection domain is *statically* bound to permissions by the security policy, it is sometimes possible to partially evaluate the conditions on the abstract edges. In particular, an edge labelled $\gamma \vdash P$ can be removed when, for some $D \in \gamma$ with static permissions, P is not granted to D.

Definition 3.5 *Let Perm be a security policy. We define:*

$$\begin{aligned} Perm &\models \quad true \\ Perm &\models (\gamma \vdash P) \iff \gamma \vdash_{Perm} P \\ Perm &\models \varphi \wedge \varphi' \iff Perm \models \varphi \text{ and } Perm \models \varphi' \\ Perm &\not\models \quad \varphi \iff \text{it is not the case that } Perm \models \varphi \end{aligned}$$

The following theorem states that our analysis is sound w.r.t. the operational semantics of CFGs: for each execution trace in G, there exists a corresponding path in $G^\sharp$ mimicking the evolution of the access control contexts.

Theorem 3.6 (Soundness) *Let* $\langle G, Perm\rangle \rhd \tau = [n_\varepsilon] \rhd \cdots \rhd \langle \sigma : n, \chi\rangle$. *Then, there exists a path* $\pi \in \Pi(\langle n, \Gamma(\sigma) \cup Dom(n), \chi\rangle)$ *such that* $\Phi(\pi) = \Phi(\tau)$.

The next theorem states that our analysis is also complete: for each path in $G^\sharp$ and security policy *Perm* that satisfies its reachability condition, there exists a corresponding execution trace in $\langle G, Perm\rangle$. This fact should not seem bizarre: indeed, completeness is only up to the precision of the CFG, which is a safe, approximated model of the analyzed program.

Theorem 3.7 (Completeness) *Let* $\pi \in \Pi(n\gamma\chi)$ *and Perm be such that* $Perm \models \Phi(\pi)$. *Then, there exist* τ, σ *such that* $\langle G, Perm\rangle \rhd \tau \rhd \langle \sigma : n, \chi\rangle$, $\gamma = \Gamma(\sigma) \cup Dom(n)$ *and* $\Phi(\tau \rhd \langle \sigma : n, \chi\rangle) = \Phi(\pi)$.

Our analysis supports a form of incremental computation, though not in a fully compositional way. This is particularly useful for *dynamic linking* of code – the mechanism which allows a program to be extended at run-time. Our program model does not directly support this feature: so we require the CFG construction algorithm to correctly link the relevant CFGs, e.g. as in [24]. Indeed, this operation cannot be performed by looking at the CFGs alone, because CFGs do not carry enough information to restrict the set of targets of dynamically dispatched method invocations.

We briefly show how the incremental computation of the analysis is performed. Assume we have a computed $G^\sharp$, when the CFG G' is loaded. The CFG construction algorithm singles out the set $\tilde{E}$ of *resolved edges* between G and G', i.e. those call edges $n \longrightarrow m$ such that n, m do not belong both to the same CFG. The linked graph $G \bowtie_{\tilde{E}} G'$ contains the nodes and the edges from both G and G', plus the resolved edges $\tilde{E}$.

To obtain $(G \bowtie_{\tilde{E}} G')^\sharp$ from $G^\sharp$, it suffices to compute the closure of the set of rules in Table 2, starting from the set of resolved edges and from the entry point of G'. Note that adding new executable traces to a CFG never affects the analysis of the old ones.

4 Program Transformations

In this section we show that our static analysis provides us with an effective basis for several code optimizations. This is not a trivial task, because performing interprocedural optimizations in the presence of stack inspection may break security. Indeed, stack inspection deeply relies on the structure of the call stack, which may be altered by such optimizations.

All the program transformations below have to be revalidated every time that the security policy is updated. This form of *dynamic deoptimization* is common practice in the presence of just-in-time compilers, e.g. the Java HotSpot Compiler [25]. Again, note that our analysis need not be recomputed.

4.1 Elimination of the redundant checks

The first application of our analysis detects the redundant checks occurring in a program, i.e. those which always pass, regardless of the execution trace.

Definition 4.1 *Let $\ell(n) =$* `check`*(P). We say that n is* redundant *w.r.t. a security policy Perm when, for each call stack σ:*

$$\langle G, Perm\rangle \rhd \sigma : n \implies \Gamma(\sigma : n) \vdash_{Perm} P$$

The following theorem states conditions to recognize redundant checks, so enabling the compiler to safely remove them from the code. Actually, redundant checks can only be *disabled* in the presence of dynamic linking, because loading a new method may add new traces where the permission is no longer granted.

Theorem 4.2 *A check node n is redundant w.r.t. a security policy Perm if and only if $Perm \not\models \Phi(n\gamma\lightning)$ for all contexts γ.*

The reachability condition $\Phi(n\gamma\not\downarrow)$ can be computed during the costruction of the abstract graph. This requires accumulating the traversability conditions on the edges leading to abstract check nodes.

4.2 Dead code elimination

Dead code elimination is a program transformation which allows the compiler to discard unreachable or useless pieces of code. This optimizazion reduces both the size of the generated bytecode and the total application running time (e.g. when code has to be downloaded from the network).

The following theorem allows to detect (and remove) those methods which cannot be reached due to security restrictions:

Theorem 4.3 *Let $n = \varepsilon(\mu(n))$. If $Perm \not\models \Phi(n\gamma)$ for all $n\gamma \in G^\sharp$, then there are no σ and $m \in \mu(n)$ such that $\langle G, Perm\rangle \rhd \sigma : m$.*

4.3 Method inlining

Method inlining is an optimization that replaces a method invocation with a copy of the called method code. As a side effect, the protection domain of the inlined method is ignored when performing stack inspection. Our analysis gives us the means to compute the set of method invocations that can be safely inlined. Intuitively, a method invocation can be inlined if the outcome of the security checks is not affected by ignoring the protection domain of the inlined method.

We adopt the so-called *original version inlining* approach [16], which always considers the original version of the callee and the current version of the caller when performing inlinings. This can be obtained by duplicating the original code of the inlined method. Let $\dot{n}$ be the node candidate for inlining, and $\dot{n} \longrightarrow n'$. We assume that the method invocation represented by $\dot{n}$ can be statically dispatched, i.e. it has exactly one callee, represented by $\mu(n')$. We also require that the protection domain of $\mu(n')$ is *isolated* in the CFG, i.e. its name is different from the other domain names (it suffices to assign to $Dom(n')$ a fresh name).

We formally specify in definition 4.4 when a method invocation can be safely inlined. The condition (2a) below guarantees static dispatching of $\dot{n}$, as well as that $\dot{n}$ is not a recursive call (otherwise inlining makes little sense). The condition (2b) says that there is a single callee: this is enforced by the original version inlining approach. The condition (2c) ensures that the protection domain of $\dot{n}$ is isolated. These conditions, apart from $\dot{n}$ being not recursive, can easily be satisfied, as noted above. The key condition is (2d): it guarantees that, for all the permissions that can be possibly checked after the inlined call,

the security policy agrees on the protection domains of $\dot{n}$ and n'.

Definition 4.4 *We say that* $\dot{n} \longrightarrow n'$ *is* inlineable *in* G *w.r.t. Perm iff:*

$$\dot{n} \longrightarrow n \implies n = n' \wedge n \notin \mu(\dot{n}) \tag{2a}$$
$$n \longrightarrow n' \implies n = \dot{n} \tag{2b}$$
$$n \notin \mu(n') \implies Dom(n) \neq Dom(n') \tag{2c}$$
$$\left.\begin{array}{r} \dot{n}\gamma \rightrightarrows m\gamma' \\ \ell(m) = \texttt{check}(P) \\ Dom(n') \in \gamma' \end{array}\right\} \implies P \in Perm(\dot{n}) \iff P \in Perm(n') \tag{2d}$$

hold for all $n, m, \gamma, \gamma', \varphi$. *We write* $\dot{n}\gamma \rightrightarrows n'\gamma'\chi'$ *when:*

$$\exists \pi, n'', \gamma''.\ \dot{n}\gamma \longrightarrow n''\gamma'' \wedge \pi \in \Pi(n''\gamma'', n'\gamma'\chi')$$

Next, we define the effect of the method inlining transformation on CFGs. Instead of substituting $\dot{n}$ for $\mu(n')$ and adjusting the edges accordingly, we equivalently operate on the semantics of the transformed CFG. The effect of the inlining of $\dot{n}$ on call stacks is specified by the function $inl_{\dot{n}}$ in Table 3. Given a state σ, $inl_{\dot{n}}(\sigma)$ is obtained by removing all the occurrences of $\dot{n}$ in σ (except when $\dot{n}$ is in top position). The operational semantics of a CFG after the inlining of $\dot{n}$ is defined by the transition relation $\rhd^{\dot{n}}$ in Table 3. For instance, the rules $\rhd^{\dot{n}}_{call1}$ and $\rhd^{\dot{n}}_{call2}$ state, respectively, that a method invocation proceeds as usual when the calling node is not $\dot{n}$, otherwise $\dot{n}$ is removed from the call stack.

Definition 4.5 *Let* $G = \langle N, E, Dom_G, Priv_G \rangle$. *The* $\dot{n}$-inlined version of G *is* $\dot{G} = \langle N, E, Dom_{\dot{G}}, Priv_{\dot{G}} \rangle$, *where:*

$$Priv_{\dot{G}}(n) = \begin{cases} Priv_G(\dot{n}) & \text{if } \dot{n} \longrightarrow \mu(n) \\ Priv_G(n) & \text{otherwise} \end{cases} \qquad Dom_{\dot{G}}(n) = \begin{cases} Dom_G(\dot{n}) & \text{if } \dot{n} \longrightarrow \mu(n) \\ Dom_G(n) & \text{otherwise} \end{cases}$$

The following theorem states the correctness of method inlining: each trace in the original CFG corresponds to a trace in the $\dot{n}$-inlined version of the CFG.

Theorem 4.6 *If* $\dot{n}$ *is inlineable in* G *w.r.t. Perm, then:*

$$\langle G, Perm \rangle \rhd \tau \quad \iff \quad \langle \dot{G}, Perm \rangle \rhd^{\dot{n}} inl_{\dot{n}}(\tau)$$

$$\frac{}{inl_{\dot n}([]) = []}\ [inl_1] \qquad \frac{inl_{\dot n}(\sigma) = \dot\sigma \quad top(\sigma) \neq \dot n}{inl_{\dot n}(\sigma : n') = \dot\sigma : n'}\ [inl_2] \qquad \frac{inl_{\dot n}(\sigma) = \dot\sigma}{inl_{\dot n}(\sigma : \dot n : n') = \dot\sigma : n'}\ [inl_3]$$

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n' \quad n \neq \dot n}{\sigma : n \rhd^{\dot n} \sigma : n : n'}\ [\rhd_{call1}] \qquad \frac{\ell(\dot n) = \texttt{call} \quad \dot n \longrightarrow n'}{\sigma : \dot n \rhd^{\dot n} \sigma : n'}\ [\rhd_{call2}]$$

$$\frac{\ell(n') = \texttt{return} \quad n \dashrightarrow m \quad \dot n \not\longrightarrow \mu(n')}{\sigma : n : n' \rhd^{\dot n} \sigma : m}\ [\rhd_{return1}] \qquad \frac{n \not\dashrightarrow_{\lightning} \quad \dot n \not\longrightarrow \mu(n')}{\sigma : n\lightning \rhd^{\dot n} \sigma\lightning}\ [\rhd_{propagate1}]$$

$$\frac{\ell(n') = \texttt{return} \quad \dot n \dashrightarrow m \quad \dot n \longrightarrow \mu(n')}{\sigma : n' \rhd^{\dot n} \sigma : m}\ [\rhd_{return2}] \qquad \frac{n \not\dashrightarrow_{\lightning} \quad \dot n \longrightarrow \mu(n')}{\sigma : n\lightning \rhd^{\dot n} \sigma : \dot n\lightning}\ [\rhd_{propagate2}]$$

Table 3
Effect of inlining $\dot n$ on call stacks (top) and transitions (bottom).

5 Conclusions and related work

We have developed a technique to perform program transformations in the presence of stack inspection and dynamic security policies. The technique relies on the definition of our Security Context Analysis. The analysis is sound and complete with respect to the control flow graphs derived from the bytecode (recall from Section 2 that these graphs safely approximate the actual behavior). Our analysis makes various optimizations possible. We focussed here on elimination of redundant checks and of dead code, and on method inlining. It is worthwhile noting that our analysis can take advantage of the control flow graphs generated by the just-in-time optimizers, e.g. the HotSpot compilers embedded in the latest Java Virtual Machines [25]. This would also make our technique directly exploitable by these tools, e.g. to produce larger methods by inlining, so allowing for further optimizations.

Many authors advocated the use of static techniques in order to understand and optimize stack inspection, among them ourselves [2,3,4,5]. As far as we can tell, our current proposal is the first one that can deal with dynamic security policies.

Besson, Jensen, Le Mètayer and Thorn [7] formalize classes of security properties through a linear-time temporal logic. They show that a large class of policies (including stack inspection) can be expressed in this formalism. Model checking is then used to prove that local security checks enforce a given global security policy. Their verification method is based on the translation from linear-time temporal formulae to deterministic finite-state automata, and

it can be used to optimize stack inspection. Based on the same model, [6] develops a static analysis that computes, for each method, the set of its *secure calling contexts* with respect to a given global property. When a method is invoked from a secure calling context, its execution never violates the global property. For some optimizations, e.g. for method inlining, this technique is even too powerful, as the information about calling contexts is unnecessary.

Esparza, Kučera and Schwoon [11] formalize stack inspection in terms of model checking pushdown systems. Obdržálek [20] uses the same technique to accurately model Java exception handling. A suitable combination of the two will then be an alternative approach to ours. Since our model is specifically tailored on stack inspection, we think that our analysis may be implemented and exploited more efficiently than a general method such as model checking pushdown systems.

Exploiting the access control logic of [1], Wallach, Appel and Felten [28] propose an alternative semantics of eager stack inspection, called *security-passing style*. This technique consists of tracking the security state of an execution as an additional parameter of each method invocation. This allows for interprocedural compiler optimizations that do not interfere with stack inspection (at the cost of more expensive method calls).

Pottier, Skalka and Smith [21] address the problem of stack inspection in λ_{sec}, a typed lambda calculus enriched with primitive constructs for enforcing security checks and managing permissions, and no exception handling. Stack inspection never fails on a well typed program, because the set of permissions granted at run-time always includes the security context. This analysis supports all-or-nothing optimizations that remove the security manager when all the checks are redundant. Instead, we can single out and remove individual redundant checks.

Fournet and Gordon [12] investigate the problem of establishing the correctness of program transformations in the presence of stack inspection. They present an equational theory, together with a coinductive proof technique, for the λ_{sec} calculus. They study how stack inspection affects program behavior, proving that certain function inlinings and tail-call eliminations are correct. The equational theory is used to reason on the (somewhat limited) security properties actually guaranteed by stack inspection. Here, we are more concerned with efficient (semantically-based) optimization procedures to be used on the field, rather than with a general reasoning framework. Indeed, it is unclear how to (mechanically) derive a procedure (e.g. a confluent terminating rewriting system) to ensure correctness of program transformations under security constraints.

Koved, Pistoia and Kershenbaum [17] address the problem of computing

the set of permissions a class needs in order to execute without throwing security exceptions. Also this analysis suffers from allowing only all-or-nothing optimizations, as in [21]. The analysis is built over *access rights invocation graphs*. These flow graphs are context-sensitive: each node is associated also with its *calling context*, i.e. with its target method, receiver and parameters values. In this way, the analysis in [17] can deal with parametric permissions and multi-threading. Our approach can gain precision through the exploitation of these graphs. We plan to study this issue in future work.

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 1993.

[2] M. Bartoletti, P. Degano, and G. L. Ferrari. Static analysis for stack inspection. In *Electronic Notes in Theoretical Computer Science*, 2001.

[3] M. Bartoletti, P. Degano, and G. L. Ferrari. Security-aware program transformations. In *Proc. 8th Italian Conference on Theoretical Computer Science*, 2003.

[4] M. Bartoletti, P. Degano, and G. L. Ferrari. Static analysis for eager stack inspection. In *Workshop on Formal Techniques for Java-like Programs*, 2003.

[5] M. Bartoletti, P. Degano, and G. L. Ferrari. Stack inspection and secure program transformations. To appear in International Journal of Information Security, 2004.

[6] F. Besson, T. de Grenier de Latour, and T. Jensen. Secure calling contexts for stack inspection. In *Proc. 4th Conference on Principles and Practice of Declarative Programming*, 2002.

[7] F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 2001.

[8] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Workshop on Program Analysis For Software Tools and Engineering*, 1999.

[9] J. Clemens and M. Felleisen. A tail-recursive semantics for stack inspections. In P. Degano, editor, *Proc. 12th European Symposium on Programming*, 2003.

[10] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, 2000.

[11] J. Esparza, A. Kučera, and S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. In *Proc. 4th International Symposium on Theoretical Aspects of Computer Software*, 2001.

[12] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems*, 2003.

[13] L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley, 1999.

[14] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 2001.

[15] G. Karjoth. An operational semantics for Java 2 access control. In *Proc. 13th Computer Security Foundations Workshop*, 2000.

[16] O. Kaser and C. R. Ramakrishnan. Evaluating inlining techniques. *Computer Languages*, 1998.

[17] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *Proc. 17th ACM conference on Object-oriented Programming, Systems, Languages, and Applications*, 2002.

[18] C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers. User authentication and authorization in the Java platform. In *Proc. 15th Annual Computer Security Application Reference*, 1999.

[19] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[20] J. Obdržálek. Model checking java using pushdown systems. In *Workshop on Formal Techniques for Java-like Programs*, 2002.

[21] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. In *Proc. 10th European Symposium on Programming*, 2001.

[22] V. Razmov. Security in untrusted code environments: Missing pieces of the puzzle. Dept. Computer Science and Engineering, Univ. of Washington, 2002.

[23] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *Software Engineering*, 2000.

[24] A. Souter and L. Pollack. Incremental call graph reanalysis for object-oriented software maintenance. In *IEEE International Conference on Software Maintenance*, 2001.

[25] Sun Microsystems. *The Java HotSpot Virtual Machine (White Paper)*.

[26] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proc. 15th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2000.

[27] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. 15th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2000.

[28] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM TOSEM*, 2001.

A Proofs

Definition A.1 *Consider a trace τ on the following form:*

$$\sigma_0\,\chi_0\ \rhd_{\ell_0}\ \sigma_1\,\chi_1\ \rhd_{\ell_1}\ \cdots\ \rhd_{\ell_{k-1}}\ \sigma_k\,\chi_k$$

For each $i, j \in 0..k$, we write $i \rightleftarrows_\tau j$ iff:

$$\ell_i = \mathit{call}\ \ \wedge\ \ i = \max\{\, h \in 0..j-1 \mid w(\tau, h, j) = 1 \,\}$$

For each $0 \le i \le j \le k$, the weight of the trace $\tau[i..j]$ is defined as follows:

$$w(\tau, i, j)\ \ =\ \ \sum_{h=i}^{j-1} w(\ell_h)$$

where the weight of a transition with label ℓ is defined as:

$$w(\ell) \;=\; \begin{cases} 1 & \text{if } \ell = \mathit{call} \\ -1 & \text{if } \ell = \mathit{return} \text{ or } \ell = \mathit{propagate} \\ 0 & \text{otherwise} \end{cases}$$

Lemma A.2 *Let $\tau = [n_\varepsilon] \rhd \cdots \rhd \langle \sigma : n : n', \chi \rangle$ be a trace of length k. Then:*

$$n \longrightarrow \mu(n') \tag{A.1a}$$
$$\exists i \in 0..k-1.\ \ \tau[i] = \sigma : n \ \wedge\ i \rightleftarrows_\tau k \tag{A.1b}$$

Proof. We proceed by induction on the length of the trace. The base case $k = 0$ holds trivially, because $\sigma_0 = [n_\varepsilon]$ does not satisfy the premises of the lemma. For the inductive case, assume (A.1a) and (A.1b) are true for all traces of length lower than k. Case analysis on the rule used to deduce $\sigma_{k-1}\chi_{k-1} \rhd \sigma_k\chi_k$ gives:

- case [*call*]:
$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n'}{\sigma : n \rhd \sigma : n : n'}$$
Here (A.1a) follows by the fact that $n \longrightarrow n'$, and the index i which satisfies (A.1b) is just $k-1$.
- case [*return*]:
$$\frac{\ell(m) = \texttt{return} \quad m' \dashrightarrow n'}{\sigma : n : m' : m \rhd \sigma : n : n'}$$
By the induction hypothesis, we have that:
$$\exists j \in 0..k-2.\ \sigma_j\chi_j = \langle \sigma : n : m', \mathit{false} \rangle \tag{A.2}$$
Since any derivation for $\sigma : n : m'$ requires at least one step, $j > 0$. Then, the induction hypothesis on $j-1$ gives $n \longrightarrow \mu(m')$, and:
$$\exists i \in 0..j-1.\ \sigma_i\chi_i = \langle \sigma : n, \mathit{false} \rangle \tag{A.3}$$
Since $m' \dashrightarrow n'$, it follows that $\mu(m') = \mu(n')$. Then, $n \longrightarrow \mu(m')$ implies $n \longrightarrow \mu(n')$. This proves (A.1a), and (A.1b) is satisfied by the index i in (A.3).
- case [*pass*]:
$$\frac{\ell(m') = \texttt{check}(P) \quad \Gamma(\sigma : n : m') \vdash_{\mathit{Perm}} P \quad m' \dashrightarrow n'}{\sigma : n : m' \rhd \sigma : n : n'}$$

By the induction hypothesis, $n \longrightarrow \mu(m')$, and:

$$\exists i \in 0..k-2.\ \sigma_i\chi_i = \langle \sigma : n, \mathit{false} \rangle \tag{A.4}$$

As in the previous case, $\mu(m') = \mu(m)$ implies that $n \longrightarrow \mu(n')$, which proves (A.1a). The index i given by (A.4) satisfies (A.1b).

- case [*fail*]:

$$\frac{\ell(n') = \texttt{check}(P) \quad \Gamma(\sigma : n : n') \not\vdash_{Perm} P}{\sigma : n : n' \rhd \sigma : n : n' \lightning}$$

Here (A.1a) and (A.1b) follow directly by the induction hypothesis.

- case [*catch*]:

$$\frac{m' \dashrightarrow_{\lightning} n'}{\sigma : n : m' \lightning \rhd \sigma : n : n'}$$

By the induction hypothesis, we have $n \longrightarrow \mu(m')$, and:

$$\exists i \in 0..k-2.\ \sigma_i\chi_i = \langle \sigma : n, \mathit{false} \rangle \tag{A.5}$$

Since $m' \dashrightarrow_{\lightning} n'$, then $\mu(m') = \mu(n')$. So, $n \longrightarrow \mu(m')$ implies $n \longrightarrow \mu(n')$, which proves (A.1a). Equation (A.1b) is satisfied by the index i in (A.5).

- case [*propagate*]:

$$\frac{m' \not\dashrightarrow_{\lightning}}{\sigma : n : n' : m' \lightning \rhd \sigma : n : n' \lightning}$$

By the induction hypothesis, we have that:

$$\exists j \in 0..k-2.\ \sigma_j\chi_j = \langle \sigma : n : n', \mathit{false} \rangle \tag{A.6}$$

Since any derivation for $\sigma : n : n'$ requires at least one step, $j > 0$. Then, the induction hypothesis on $j-1$ gives $n \longrightarrow \mu(n')$, and:

$$\exists i \in 0..j-1.\ \sigma_i\chi_i = \langle \sigma : n, \mathit{false} \rangle$$

Definition A.3 *Let $\tau = \sigma_0\chi_0 \rhd_{\ell_1} \cdots \rhd_{\ell_k} \sigma_k\chi_k$. We define the* flattening *of τ as $\flat(\tau) = \flat(\tau, 0, k)$, where, for $0 \le i \le j \le k$:*

$$\flat(\tau, i, j) = \begin{cases} \sigma_j\chi_j & \text{if } i = j \\ \flat(\tau, i, h) \xrightarrow{\Phi(\tau[h..j])} \sigma_j\chi_j & \text{if } i \le h \rightleftarrows_\tau j-1, w(\ell_j) = -1 \\ \flat(\tau, i, j-1) \xrightarrow{\Phi(\tau[j-1..j])} \sigma_j\chi_j & \text{otherwise} \end{cases}$$

Lemma A.4 *Let τ be a trace of length k, with $\flat(\tau) = \sigma_0\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_h} \sigma_h\chi_h$.*

$$h \leq k \tag{A.7a}$$
$$\sigma_0\chi_0 = \tau[0],\ \sigma_h\chi_h = \tau[k] \tag{A.7b}$$
$$\Phi(\tau) = \textstyle\bigwedge_{i=0..h} \varphi_i \tag{A.7c}$$

Proof. We won't go through the details of the proof for (A.7a) and (A.7b), because they are immediate from definition A.3. The proof of (A.7c) is by induction on the length of τ. If $k = 0$, then $\flat(\tau) = \sigma_0\chi_0$, and by convention, $\bigwedge \emptyset = \mathit{true} = \Phi(\tau)$.

For the inductive case, let $k > 0$. We proceed by case analysis on the rule applicable to compute $\flat(\tau, 0, k)$. The first rule cannot be applied, because $k > 0$. The second rule, requiring $h \rightleftarrows_\tau k{-}1$, $\sigma_{k-1}\chi_{k-1} \rhd_\ell \sigma_k\chi_k$ and $w(\ell) = -1$, gives:

$$\flat(\tau) \;=\; \flat(\tau, 0, h) \xrightarrow{\Phi(\tau[h..k])} \sigma_k\chi_k$$

By the induction hypothesis, $\Phi(\tau[0..h]) = \bigwedge_{i=0..h-1} \varphi_i$. Then:

$$\Phi(\tau) \;=\; \Phi(\tau[0..h]) \wedge \Phi(\tau[h..k]) \;=\; (\textstyle\bigwedge_{i=0..h-1} \varphi_i) \wedge \varphi_h \;=\; \bigwedge_{i=0..h} \varphi_i$$

The third rule in the definition of $\flat$ is applicable in any other case, and it gives:

$$\flat(\tau) \;=\; \flat(\tau, 0, k-1) \xrightarrow{\Phi(\tau[k-1..k])} \sigma_k\chi_k$$

By the induction hypothesis, $\Phi(\tau[0..k-1]) = \bigwedge_{i=0..h-1} \varphi_i$. Then:

$$\Phi(\tau) \;=\; \Phi(\tau[0..k-1]) \wedge \Phi(\tau[k-1..k]) \;=\; (\textstyle\bigwedge_{i=0..h-1} \varphi_i) \wedge \varphi_h \;=\; \bigwedge_{i=0..h} \varphi_i$$

Definition A.5 *We say that $\tau = \sigma_0\chi_0 \rhd \cdots \rhd \sigma_k\chi_k$ is* positive *when:*

$$\forall i \in 1..k.\ w(\tau, 0, i) \geq 0$$

Similarly, we say that τ is strictly positive *when $>$ holds in place of $\geq$.*

Lemma A.6 *Let τ be a trace of length k. Then:*

$$0 \rightleftarrows_\tau k \quad\Longrightarrow\quad \flat(\tau) = \tau[0] \rightarrow \flat(\tau, 1, k) \tag{A.8}$$

Proof. For $k = 0$, we have that $0 \not\rightleftarrows_\tau 0$, therefore (A.8) holds trivially. For $k > 0$, we prove (A.8) as a corollary of the following, more general, result.

Let τ be a strictly positive trace, of length $k > 0$. Then:

$$\flat(\tau) = \tau[0] \xrightarrow{\Phi(\tau[0..1])} \flat(\tau, 1, k) \tag{A.9}$$

We prove (A.9) by induction on the length of τ. For the base case, if $k = 0$ there is nothing to prove. For the inductive case, let $k > 0$. We proceed by case analysis on the rule applicable to compute $\flat(\tau, 0, k)$. The first rule cannot be applied, because $k > 0$. The second rule, requiring $h \rightleftarrows_\tau k-1$, $\tau[k-1] \rhd_\ell \tau[k]$ and $w(\ell) = -1$, gives:

$$\flat(\tau) \;=\; \flat(\tau, 0, h) \xrightarrow{\Phi(\tau[h..k])} \tau[k] \tag{A.10}$$

Consider $\tau[0..h]$. Since $h \rightleftarrows_\tau k-1$ and $w(\tau[k-1] \rhd \tau[k]) = -1$, if it were $h = 0$ then $w(\tau, 0, k) = 0$, which would contradict our assumption about τ being strictly positive. Therefore, $h > 0$. Since any prefix of a strictly positive trace is strictly positive itself, we can apply the induction hypothesis on $\tau[0..h]$ to obtain:

$$\flat(\tau, 0, h) \;=\; \tau[0] \xrightarrow{\Phi(\tau[0..1])} \flat(\tau, 1, h) \tag{A.11}$$

Now consider $\tau[1..k]$. Since $h \geq 1$, the second rule in the definition of $\flat$ gives:

$$\flat(\tau, 1, k) \;=\; \flat(\tau, 1, h) \xrightarrow{\Phi(\tau[h..k])} \tau[k] \tag{A.12}$$

Putting together (A.10), (A.11) and (A.12), we obtain:

$$\begin{aligned} \flat(\tau) \;&=\; \flat(\tau, 0, h) \xrightarrow{\Phi(\tau[h..k])} \tau[k] \\ &=\; \tau[0] \xrightarrow{\Phi(\tau[0..1])} \flat(\tau, 1, h) \xrightarrow{\Phi(\tau[h..k])} \tau[k] \\ &=\; \tau[0] \xrightarrow{\Phi(\tau[0..1])} \flat(\tau, 1, k) \end{aligned}$$

The third rule in the definition of $\flat$ is applicable in any other case, and it states:

$$\flat(\tau) \;=\; \flat(\tau, 0, k-1) \xrightarrow{\Phi(\tau[k-1..k])} \tau[k] \tag{A.13}$$

If $k = 1$, by the first rule in the definition of $\flat$, it follows that $\flat(\tau, 0, k-1) = \tau[0]$ and $\flat(\tau, 1, k) = \tau[k]$, so we are done. Otherwise, if $k > 1$ we can apply the induction hypothesis on $\tau[0..k-1]$ (a prefix of a strictly positive trace) to obtain:

$$\flat(\tau, 0, k-1) \;=\; \tau[0] \xrightarrow{\Phi(\tau[0..1])} \flat(\tau, 1, k-1) \tag{A.14}$$

Now consider $\tau[1..k]$. The second rule in the definition of $\flat$ is not applicable to compute $\flat(\tau[1..k])$ – otherwise it would have been applied also to compute $\flat(\tau[0..k])$. Thus, the third rule gives:

$$\flat(\tau, 1, k) \;=\; \flat(\tau, 1, k-1) \xrightarrow{\Phi(\tau[k-1..k])} \tau[k] \tag{A.15}$$

Putting together (A.13), (A.14) and (A.15), we obtain:

$$\begin{aligned}\flat(\tau) &= \flat(\tau,0,k-1) \xrightarrow{\Phi(\tau[k-1..k])} \tau[k] \\ &= \tau[0] \xrightarrow{\Phi(\tau[0..1])} \flat(\tau,1,k-1) \xrightarrow{\Phi(\tau[k-1..k])} \tau[k] \\ &= \tau[0] \xrightarrow{\Phi(\tau[0..1])} \flat(\tau,1,k)\end{aligned}$$

This concludes the proof of (A.9). To prove (A.8), we just need to show that $0 \rightleftarrows_\tau k$ implies that τ is strictly positive – indeed, $\Phi(\tau[0..1]) = \mathit{true}$ follows directly by definition 3.4. By definition A.1 we have that, for each $j \in 1..k$, $w(\tau,j,k) < 1$. Now, let $i \in 1..k$. Then:

$$w(\tau,0,i) = w(\tau,0,k) - w(\tau,i,k) > 1-1 = 0$$

Lemma A.7 *Let τ be a positive trace on $\langle G, Perm\rangle$. Let:*

$$\flat(\tau) = (\sigma_0 : n_0)\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_l} (\sigma_l : n_l)\chi_l$$

Let $\gamma_0 = \Gamma(\sigma_0) \cup Dom(n_0)$. If $n_0\gamma_0\chi_0 \in N^\sharp$, then there exists a path:

$$\pi = n_0\gamma_0\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_l} n_l\gamma_l\chi_l$$

in $G^\sharp$ such that $w(\pi) = w(\tau)$, and $\gamma_i = \Gamma(\sigma_i) \cup Dom(n_i)$ for each $i = 1..l$.

Proof. Assume that τ consists of k steps: we proceed by induction on k. The base case $k = 0$ requires $\tau = (\sigma_0 : n_0)\chi_0$, which is positive. Since $n_0\gamma_0\chi_0 \in N^\sharp$ by hypothesis, the path π is just the single node $n_0\gamma_0\chi_0$, and $w(\pi) = 0 = w(\tau)$.

For the inductive case, let $k > 0$, $n_0\gamma_0\chi_0 \in N^\sharp$. We proceed by case analysis on the rule applicable to compute $\flat(\tau,0,k)$. The first rule cannot be applied, because $k > 0$. The second rule requires $h \rightleftarrows_\tau k-1$, $w(\ell_k) = -1$, and it gives:

$$\flat(\tau) = \flat(\tau,0,h) \xrightarrow{\Phi(\tau[h..k])} \sigma_k\chi_k$$

Consider first the trace $\tau' = \tau[0..h]$. By definition A.5 it follows that each prefix of a positive trace is positive: therefore, τ' is positive. Since the length of τ' is $h < k-1 < k$, and $n_0\gamma_0\chi_0 \in N^\sharp$, then the induction hypothesis gives a path:

$$\pi' = n_0\gamma_0\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_{l-1}} n_{l-1}\gamma_{l-1}\chi_{l-1}$$

with $w(\pi') = w(\tau')$ and $\gamma_i = \Gamma(\sigma_i) \cup Dom(n_i)$ for $i = 1..l-1$.

Next, consider $\tau'' = \tau[h..k-1]$. Since $h \rightleftarrows_\tau k-1$, by definition A.1 we have that, for each $j \in h..k-1$, $w(\tau,j,k-1) \leq 1$. Now, let $i \in h..k-1$.

Then:

$$w(\tau,h,i) \;=\; w(\tau,h,k-1) - w(\tau,i,k-1) \;\geq\; 1-1 \;=\; 0$$

It follows that τ'' is positive. Let:

$$\flat(\tau'') \;=\; (\sigma'_0 : n'_0)\chi'_0 \xrightarrow{\varphi'_1} \cdots \xrightarrow{\varphi'_p} (\sigma'_p : n'_p)\chi'_p$$

By lemma A.4, we have that:

$$\begin{aligned}(\sigma'_0 : n'_0)\chi'_0 = \flat(\tau'')[0] \;&=\; \tau''[0] \;=\; \tau[h] \\ = \tau'[h] \;&=\; \flat(\tau')[l-1] \;=\; (\sigma_{l-1} : n_{l-1})\chi_{l-1}\end{aligned} \tag{A.16}$$

Let $\gamma'_0 = \Gamma(\sigma'_0) \cup \mathit{Dom}(n'_0)$. By (A.16), $\gamma'_0 = \gamma_{l-1} = \Gamma(\sigma_{l-1}) \cup \mathit{Dom}(n_{l-1})$. It follows that $n'_0\gamma'_0\chi'_0 = n_{l-1}\gamma_{l-1}\chi_{l-1} \in N^\sharp$. Since the length of τ'' is $k-1-h < k$, we can apply the induction hypothesis to obtain a path:

$$\pi'' \;=\; n'_0\gamma'_0\chi'_0 \xrightarrow{\varphi'_1} \cdots \xrightarrow{\varphi'_p} n'_p\gamma'_p\chi'_p$$

with $w(\pi'') = w(\tau'') = w(\tau,h,k-1) = 1$ (because $h \rightleftarrows_\tau k-1$), and $\gamma'_i = \Gamma(\sigma'_i) \cup \mathit{Dom}(n'_i)$ for $i = 1..p$.

By lemma A.4 and definition 3.3, $\langle \sigma'_0 : n'_0, \chi'_0 \rangle = \tau[h] \rhd_{\mathit{call}} \tau[h+1]$. So, $\chi'_0 = \mathit{false}$, and $\tau[h+1]$ is on the form $\sigma'_0 : n'_0 : n$, for some $n \in N$ such that $n'_0 \longrightarrow n$. Since $h \rightleftarrows_\tau k-1$, by lemma A.6 it follows that $\flat(\tau'') = \tau[h] \rightarrow \flat(\tau, h+1, k-1)$. Lemma A.4 also ensures that the first element of $\flat(\tau, h+1, k-1)$ is just $\tau[h+1]$: then, $\flat(\tau'') = [\tau[h], \tau[h+1], \ldots]$, that implies $n = n'_1$ and $\sigma'_1 = \sigma'_0 : n'_0$. Now, consider the first edge in π'', i.e. $n'_0\gamma'_0 \rightarrow n'_1\gamma'_1$. Since $w(\pi'') = 1 = w(\pi''[0..1])$ and the weight of a path cannot decrease by adding new nodes, then $w(\pi''[1..p]) = 0$. Let $\varphi = \bigwedge_{i=1..p} \varphi'_i$. Then, by definition 3.3, $n'_0\gamma'_0 \rightleftarrows_\varphi n'_p\gamma'_p\chi'_p$.

Now, we have to deal with the two possible values of the label ℓ_k. Consider first the case $\ell_k = \mathit{return}$. By lemma A.4, $\sigma'_p : n'_p = \tau[k-1]$ and $\sigma_l : n_l = \tau[k]$. Since $\rhd_{\ell_k}$ is a return transition, it follows that $\ell(n'_p) = \texttt{return}$, and $\sigma'_p : n'_p$ must be on the form $\sigma' : n' : n'_p$ for some σ' and n'. By lemma A.2, there exists an index $i \in 1..k-2$ such that $i \rightleftarrows_\tau k-1$ and $\tau[i] = \sigma' : n'$. Definition A.1 implies that, given a trace τ and an index j, there exists a *unique* index i such that $i \rightleftarrows_\tau j$. Since $h \rightleftarrows_\tau k-1$ by hypothesis, then $h = i$, and $\sigma'_p = \sigma' : n' = \tau[i] = \tau[h] = \sigma'_0 : n'_0$. Therefore, the $\rhd_{\mathit{return}}$ rule applied at step k instances to:

$$\frac{\ell(n'_p) = \texttt{return} \quad n'_0 \dashrightarrow n_l}{\sigma'_0 : n'_0 : n'_p \rhd \sigma'_0 : n_l}$$

Then, $\chi_l = \mathit{false}$ and $\gamma_l = \Gamma(\sigma_l) \cup \mathit{Dom}(n_l) = \Gamma(\sigma'_0) \cup \mathit{Dom}(n'_0) = \gamma'_0$. The $\sharp_{return}$ rule instances to:

$$\frac{\ell(n'_p) = \texttt{return} \quad n'_0 \dashrightarrow n_l \quad n'_0\gamma'_0 \rightleftarrows_\varphi n'_p\gamma'_p}{n'_0\gamma'_0 \xrightarrow{\varphi} n_l\gamma'_0}$$

Therefore, the path $\pi = \pi' \xrightarrow{\varphi} n_l\gamma_l$ is in $G^\sharp$, and:

$$w(\pi) = w(\pi') + w(n'_0\gamma'_0 \xrightarrow{\varphi} n_l\gamma'_0) = 1 + 0$$
$$w(\tau) = w(\tau') + w(\tau'') + w(\sigma'_0 : n'_0 : n'_p \rhd \sigma'_0 : n'_0) = 1 + 1 - 1$$

Next, we consider the case $\ell_k = \mathit{propagate}$. By lemma A.4, $\sigma'_p : n'_p\lightning = \tau[k-1]$ and $\sigma_l : n_l\lightning = \tau[k]$. The $\rhd_{propagate}$ rule instances to:

$$\frac{n'_p \not\dashrightarrow_\lightning}{\sigma'_p : n'_p\lightning \rhd \sigma'_p\lightning}$$

With the same arguments used above, we deduce that $\sigma'_0 : n'_0 = \sigma'_p = \sigma_l : n_l$, and $\gamma'_0 = \gamma_l$. By the $\sharp_{propagate}$ rule:

$$\frac{n'_0\gamma'_0 \rightleftarrows_\varphi n'_p\gamma'_p\lightning \quad n'_p \not\dashrightarrow_\lightning}{n'_0\gamma'_0 \xrightarrow{\varphi} n'_0\gamma'_0\lightning}$$

As above, the path $\pi = \pi' \xrightarrow{\varphi} n_l\gamma_l\lightning$ is in $G^\sharp$, and $w(\pi) = w(\tau)$.

The third rule in the definition of $\flat$ is applicable in any other case. Let $\tau' = \tau[0..k-1]$ and $\varphi = \Phi(\sigma_{k-1}\chi_{k-1} \rhd_{\ell_k} \sigma_k\chi_k)$. The rule states that:

$$\flat(\tau) \quad = \quad \flat(\tau') \xrightarrow{\varphi} \sigma_k\chi_k$$

Then, $\sigma_k\chi_k = (\sigma_l : n_l)\chi_l$. By the induction hypothesis, there exists a path:

$$\pi' \quad = \quad n_0\gamma_0\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_{l-1}} n_{l-1}\gamma_{l-1}\chi_{l-1}$$

with $w(\pi') = w(\tau')$ and $\gamma_i = \Gamma(\sigma_i) \cup \mathit{Dom}(n_i)$ for $i = 1..l-1$. By case analysis on the value of the label ℓ_k, we have:

- case $[\mathit{call}]$. By the $\rhd_{call}$ rule:

$$\frac{\ell(n_{l-1}) = \texttt{call} \quad n_{l-1} \longrightarrow n_l}{\sigma_{l-1} : n_{l-1} \rhd \sigma_{l-1} : n_{l-1} : n_l}$$

Then, $\sigma_l = \sigma_{l-1} : n_{l-1}$, and $\chi_{l-1} = \chi_l = \mathit{false}$. By definition 2.3:

$$\begin{array}{rcl} \gamma_l & = & \Gamma(\sigma_{l-1} : n_{l-1}) \cup \mathit{Dom}(n_l) \\ & = & \Gamma(\sigma_{l-1}) \uparrow n_{l-1} \cup \mathit{Dom}(n_l) \\ & = & \big(\Gamma(\sigma_{l-1}) \cup \mathit{Dom}(n_{l-1})\big) \uparrow n_{l-1} \cup \mathit{Dom}(n_l) \\ & = & \gamma_{l-1} \uparrow n_{l-1} \cup \mathit{Dom}(n_l) \end{array}$$

The $\sharp_{call}$ rule gives:

$$\frac{\ell(n_{l-1}) = \texttt{call} \quad n_{l-1} \longrightarrow n_l \quad n_{l-1}\gamma_{l-1} \in N^\sharp}{n_{l-1}\gamma_{l-1} \longrightarrow n_l\gamma_l}$$

Then, $\pi = \pi' \longrightarrow n_l\gamma_l$ is in $G^\sharp$, and $w(\pi) = w(\pi') + 1 = w(\tau') + 1 = w(\tau)$.

- case [*pass*]. By the $\rhd_{pass}$ rule:

$$\frac{\ell(n_{l-1}) = \texttt{check}(P) \quad \Gamma(\sigma_{l-1} : n_{l-1}) \vdash_{Perm} P \quad n_{l-1} \dashrightarrow n_l}{\sigma_{l-1} : n_{l-1} \rhd \sigma_{l-1} : n_l}$$

Then, $\sigma_l = \sigma_{l-1}$ and $\chi_{l-1} = \chi_l = \mathit{false}$. By definition 3.4, it follows that $\varphi = \Gamma(\sigma_{l-1} : n_{l-1}) \vdash P$. Since n_{l-1} is not privileged (constraint (1e)), then:

$$\gamma_{l-1} \quad = \quad \Gamma(\sigma_{l-1}) \cup \mathit{Dom}(n_{l-1}) \quad = \quad \Gamma(\sigma_{l-1} : n_{l-1})$$

Then, $\varphi = \gamma_{l-1} \vdash P$, and by the $\sharp_{pass}$ rule:

$$\frac{\ell(n_{l-1}) = \texttt{check}(P) \quad n_{l-1}\gamma_{l-1} \in N^\sharp \quad n_{l-1} \dashrightarrow n_l}{n_{l-1}\gamma_{l-1} \xrightarrow{\gamma_{l-1} \vdash P} n_l\gamma_{l-1}}$$

By constraint (1d), $\gamma_l = \Gamma(\sigma_l) \cup \mathit{Dom}(n_l) = \Gamma(\sigma_{l-1}) \cup \mathit{Dom}(n_{l-1}) = \gamma_{l-1}$. Then, $\pi = \pi' \xrightarrow{\gamma_{l-1} \vdash P} n_l\gamma_l$ is in $G^\sharp$, and $w(\pi) = w(\pi') + 0 = w(\tau') = w(\tau)$.

- case [*fail*]. By the $\rhd_{fail}$ rule:

$$\frac{\ell(n_{l-1}) = \texttt{check}(P) \quad \Gamma(\sigma_{l-1} : n_{l-1}) \nvdash_{Perm} P}{\sigma_{l-1} : n_{l-1} \rhd \sigma_{l-1} : n_{l-1}\lightning}$$

Similarly to the previous case, $\varphi = \Gamma(\sigma_{l-1} : n_{l-1}) \nvdash P = \gamma_{l-1} \nvdash P$, and $\sigma_l : n_l = \sigma_{l-1} : n_{l-1}$, $\gamma_l = \gamma_{l-1}$. Then, by the $\sharp_{fail}$ rule:

$$\frac{\ell(n_{l-1}) = \texttt{check}(P) \quad n_{l-1}\gamma_{l-1} \in N^\sharp}{n_{l-1}\gamma_{l-1} \xrightarrow{\gamma_{l-1} \nvdash P} n_{l-1}\gamma_{l-1}\lightning}$$

Then, $\pi = \pi' \xrightarrow{\gamma_{l-1} \not\vdash P} n_l\gamma_l\lightning$ is in $G^\sharp$, and $w(\pi) = w(\tau)$.

- case $[catch]$. By the $\rhd_{catch}$ rule:

$$\frac{n_{l-1} \dashrightarrow_\lightning n_l}{\sigma_{l-1} : n_{l-1}\lightning \rhd \sigma_{l-1} : n_l}$$

Here $\sigma_l = \sigma_{l-1}$. By constraint (1d), $\gamma_l = \gamma_{l-1}$, and, by definition 3.4, $\varphi = true$. Then, by the $\sharp_{catch}$ rule:

$$\frac{n_{l-1}\gamma_{l-1}\lightning \in N^\sharp \quad n_{l-1} \dashrightarrow_\lightning n_l}{n_{l-1}\gamma_{l-1}\lightning \rightarrow n_l\gamma_{l-1}}$$

The path $\pi = \pi' \rightarrow n_l\gamma_l$ is in $G^\sharp$, and $w(\pi) = w(\pi') + 0 = w(\tau') + 0 = w(\tau)$.

Proof of Theorem 3.6 Let $\tau = [n_\varepsilon] \rhd \cdots \rhd \langle \sigma : n, \chi\rangle$ be a trace on $\langle G, Perm\rangle$. Since τ cannot contain intermediate states on the form $[]$ or $[]\lightning$, it follows that τ is positive. Let $\flat(\tau) = [n_\varepsilon] \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_k} (\sigma : n)\chi$. Since $n_\varepsilon\emptyset \in N^\sharp$, then lemma A.7 ensures that there exists a path π in $G^\sharp$ whose last node is $\langle n, \Gamma(\sigma) \cup Dom(n), \chi\rangle$, and $\Phi(\pi) = \bigwedge_i \varphi_i = \Phi(\tau)$ by lemma A.4.

Lemma A.8 *Let $\langle G, Perm\rangle \rhd \tau$. Then:*

$$Perm \models \Phi(\tau) \tag{A.17a}$$
$$\forall Perm' \models \Phi(\tau).\ \langle G, Perm'\rangle \rhd \tau \tag{A.17b}$$

Proof. We first prove (A.17a), by induction on the length of τ. For the base case $\tau = []$ there is nothing to prove, because $\Phi([]) = true$ by convention. For the inductive case, let k be the length of τ, and consider the last step $\tau[k-1] \rhd_\ell \tau[k]$ in the trace. According to definition 3.4, if $\ell \notin \{pass, fail\}$ then $\Phi(\tau[k-1..k]) = true$: then, $Perm \models \Phi(\tau[0..k-1]) \wedge true$ follows by the induction hypothesis and by definition 3.5. If $\ell = pass$, then the last step of τ is on the form:

$$\frac{\ell(n) = \texttt{check}(P) \quad \Gamma(\sigma : n) \vdash_{Perm} P \quad n \dashrightarrow m}{\sigma : n \rhd \sigma : m}$$

By definition 3.4, $\Phi(\tau[k-1..k]) = \Gamma(\sigma : n) \vdash P$. By definition 3.5:

$$\begin{array}{rl} & Perm \models (\Gamma(\sigma : n) \vdash P) \wedge \Phi(\tau[0..k-1]) \\ \iff & \Gamma(\sigma : n) \vdash_{Perm} P \wedge Perm \models \Phi(\tau[0..k-1]) \end{array}$$

which follows by the premises of the $\rhd_{pass}$ rule and by the induction hypothesis. The case $\ell = \mathit{fail}$ is similar.

For (A.17b), let $\mathit{Perm}' \models \Phi(\tau)$. The only steps in the derivation which are sensitive to the security policy are those labelled *pass* or *fail*. Let $\ell(n) = \texttt{check}(P)$, and $\sigma : n \rhd_{pass} \sigma : m$ be a transition on $\langle G, \mathit{Perm}\rangle$. Then, $\Gamma(\sigma : n) \vdash_{\mathit{Perm}} P$. Since $\mathit{Perm} \models \Phi(\tau)$ by (A.17a), then definition 3.4 implies that $\mathit{Perm} \models \Gamma(\sigma : n) \vdash P$. Now, we have assumed that $\mathit{Perm}' \models \Phi(\tau)$, so $\mathit{Perm}' \models \Gamma(\sigma : n) \vdash P$, too. Therefore, $\Gamma(\sigma : n) \vdash_{\mathit{Perm}'} P$, which enables the transition $\sigma : n \rhd_{pass} \sigma : m$ also on $\langle G, \mathit{Perm}\rangle$. The case *fail* is treated similarly.

Lemma A.9 *Let $\pi = n_0\gamma_0\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_k} n_k\gamma_k\chi_k$ be a path in $G^\sharp$, $\mathit{Perm} \models \Phi(\pi)$ and $\langle G, \mathit{Perm}\rangle \rhd \tau' \rhd \langle \sigma_0 : n_0, \chi_0\rangle$ for some τ' and σ_0 s.t. $\gamma_0 = \Gamma(\sigma_0) \cup \mathit{Dom}(n_0)$. Then, there exists a positive trace τ such that $\langle G, \mathit{Perm}\rangle \rhd \tau' \rhd \tau$, and:*

$$\flat(\tau) \;=\; (\sigma_0 : n_0)\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_k} (\sigma_k : n_k)\chi_k \tag{A.18a}$$

Moreover, $w(\pi) = w(\tau)$, $\Phi(\pi) = \Phi(\tau)$, and, for each $i \in 1..k$:

$$\sigma_i \;=\; \begin{cases} \sigma_{i-1} : n_{i-1} & \text{if } w(\pi[i-1] \rightarrow \pi[i]) = 1 \\ \sigma_{i-1} & \text{otherwise} \end{cases} \tag{A.18b}$$

$$\gamma_i \;=\; \Gamma(\sigma_i) \cup \mathit{Dom}(n_i) \tag{A.18c}$$

Proof. By induction on the derivation of π. The base case requires $\pi = n_\varepsilon \mathit{Dom}(n_\varepsilon)$, $\tau' = []$, $\sigma_0 = []$, $n_0 = n_\varepsilon$ and $\chi_0 = \mathit{false}$. Let $\tau = [n_\varepsilon]$. Then, $\langle G, \mathit{Perm}\rangle \rhd \tau' \rhd \tau$, and the other statements of the lemma hold trivially. For the inductive case, we proceed by case analysis on the last rule used in the derivation of π. We consider only the cases $\sharp_{call}$, $\sharp_{return}$ and $\sharp_{pass}$ – the other cases can be treated similarly.

- case $[\mathit{call}]$. By the $\sharp_{call}$ rule:

$$\frac{\ell(n_{k-1}) = \texttt{call} \quad n_{k-1}\gamma_{k-1} \in N^\sharp \quad n_{k-1} \longrightarrow n_k}{n_{k-1}\gamma_{k-1} \rightarrow n_k(\gamma_{k-1} \uparrow n_{k-1}) \cup \mathit{Dom}(n_k)}$$

  Then, $\varphi_k = \mathit{true}$ and $\chi_{k-1} = \chi_k = \mathit{false}$. By the induction hypothesis, there exists a positive trace τ'' such that $\langle G, \mathit{Perm}\rangle \rhd \tau' \rhd \tau''$, $w(\tau'') = w(\pi[0..k-1])$, $\Phi(\tau'') = \Phi(\pi[0..k-1])$, and:

$$\flat(\tau'') \;=\; (\sigma_0 : n_0)\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_{k-1}} \sigma_{k-1} : n_{k-1}$$

By the $\rhd_{call}$ rule, we have:

$$\frac{\ell(n_{k-1}) = \texttt{call} \quad n_{k-1} \longrightarrow n_k}{\sigma_{k-1} : n_{k-1} \rhd \sigma_{k-1} : n_{k-1} : n_k}$$

Let $\sigma_k = \sigma_{k-1} : n_{k-1}$ and $\tau = \tau'' \rhd \sigma_k$. The third rule in definition A.3 gives:

$$\flat(\tau) \;=\; \flat(\tau'') \to \sigma_k : n_k \;=\; (\sigma_0 : n_0)\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_k} (\sigma_k : n_k)\chi_k$$

This proves (A.18a). For (A.18b), note that $\sigma_k = \sigma_{k-1} : n_k$ is coherent with $w(\pi[k-1] \to \pi[k]) = 1$. For (A.18c), the induction hypothesis and def. 2.3 give:

$$\begin{aligned}\gamma_k &= (\gamma_{k-1} \uparrow n_{k-1}) \cup Dom(n_k)\\ &= \big((\Gamma(\sigma_{k-1}) \cup Dom(n_{k-1})) \uparrow n_{k-1}\big) \cup Dom(n_k)\\ &= (\Gamma(\sigma_{k-1}) \uparrow n_{k-1}) \cup Dom(n_k)\\ &= \Gamma(\sigma_k) \cup Dom(n_k)\end{aligned}$$

To conclude the proof of this case, note that the induction hypothesis gives:

$$\begin{aligned}w(\pi) &= w(\pi[0..k-1]) + 1 = w(\tau'') + 1 = w(\tau)\\ \Phi(\pi) &= \Phi(\pi[0..k-1]) \wedge \mathit{true} = \Phi(\tau'') = \Phi(\tau)\end{aligned}$$

- case $[\mathit{return}]$. By the $\sharp_{return}$ rule, there exist n and γ such that:

$$\frac{\ell(n) = \texttt{return} \quad n_{k-1} \dashrightarrow n_k \quad n_{k-1}\gamma_{k-1} \rightleftarrows_{\varphi_k} n\gamma}{n_{k-1}\gamma_{k-1} \xrightarrow{\varphi_k} n_k\gamma_{k-1}}$$

Then, $\gamma_k = \gamma_{k-1}$ and $\chi_k = \chi_{k-1} = \mathit{false}$. By the induction hypothesis, there exists a positive trace τ_0 (of length l_0) such that $\langle G, \mathit{Perm}\rangle \rhd \tau' \rhd \tau_0$, and:

$$w(\tau_0) = w(\pi[0..k-1]) \tag{A.19a}$$
$$\Phi(\tau_0) = \Phi(\pi[0..k-1]) \tag{A.19b}$$
$$\flat(\tau_0) = (\sigma_0 : n_0)\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_{k-1}} \sigma_{k-1} : n_{k-1} \tag{A.19c}$$

Moreover, for each $i \in 1..k-1$:

$$\sigma_i = \begin{cases}\sigma_{i-1} : n_{i-1} & \text{if } w(\pi[i-1] \to \pi[i]) = 1\\ \sigma_{i-1} & \text{otherwise}\end{cases} \tag{A.19d}$$
$$\gamma_i = \Gamma(\sigma_i) \cup Dom(n_i) \tag{A.19e}$$

Consider the path π'' used to derive $n_{k-1}\gamma_{k-1} \rightleftarrows_{\varphi_k} n\gamma$. By definition 3.3, $n_{k-1} \longrightarrow \varepsilon(\mu(n))$, $\gamma = (\gamma_{k-1} \uparrow n_{k-1}) \cup Dom(n)$, and π'' is on the form $n_{k-1}\gamma_{k-1} \rightarrow \pi'$ for some $\pi' \in \Pi(\varepsilon(\mu(n))\gamma, n\gamma)$ such that $w(\pi') = 0$ and $\Phi(\pi') = \varphi_k$. By the $\rhd_{call}$ rule, we can derive:

$$\frac{\ell(n_{k-1}) = \texttt{call} \quad n_{k-1} \longrightarrow \varepsilon(\mu(n))}{\sigma_{k-1} : n_{k-1} \rhd \sigma_{k-1} : n_{k-1} : \varepsilon(\mu(n))}$$

By (A.19e), definition 2.3 and constraint (1d), we have that:

$$\begin{aligned}\gamma &= (\gamma_{k-1} \uparrow n_{k-1}) \cup Dom(n)\\ &= \big(\Gamma(\sigma_{k-1}) \cup Dom(n_{k-1})\big) \uparrow n_{k-1} \cup Dom(\varepsilon(\mu(n)))\\ &= \Gamma(\sigma_{k-1} : n_{k-1}) \cup Dom(\varepsilon(\mu(n)))\end{aligned}$$

Then, we can apply the induction hypothesis to obtain a positive trace τ_1 (of length l_1) such that $\langle G, Perm\rangle \rhd \tau' \rhd \tau_0 \rhd \tau_1$, and:

$$w(\tau_1) = w(\pi') = 0 \tag{A.20a}$$
$$\Phi(\tau_1) = \Phi(\pi') = \varphi_k \tag{A.20b}$$
$$\flat(\tau_1) = (\sigma'_0 : n'_0)\chi'_0 \xrightarrow{\varphi'_1} \cdots \xrightarrow{\varphi'_h} (\sigma'_h : n'_h)\chi'_h \tag{A.20c}$$

with $\sigma'_0 = \sigma_{k-1} : n_{k-1}$, $n'_0 = \varepsilon(\mu(n))$, $\chi'_0 = \mathit{false}$, $n'_h = n$ and $\chi'_h = \mathit{false}$. Moreover for each $i \in 1..h-1$:

$$\sigma'_i = \begin{cases} \sigma'_{i-1} : n'_{i-1} & \text{if } w(\pi'[i-1] \rightarrow \pi'[i]) = 1\\ \sigma'_{i-1} & \text{otherwise}\end{cases} \tag{A.20d}$$

Since $w(\pi') = 0$, and the weight of a path is non-decreasing, it follows that $w(\pi'[i] \rightarrow \pi'[i+1]) = 0$ for each $i \in 0..h-1$. Then, by (A.20d), $\sigma'_h = \sigma'_0 = \sigma_{k-1} : n_{k-1}$. Thus, the $\rhd_{return}$ rule gives:

$$\frac{\ell(n) = \texttt{return} \quad n_{k-1} \dashrightarrow n_k}{\sigma_{k-1} : n_{k-1} : n \rhd \sigma_{k-1} : n_k}$$

Let $\tau = \tau_0 \rhd \tau_1 \rhd \sigma_{k-1} : n_k$. We have just proved that $\langle G, Perm\rangle \rhd \tau' \rhd \tau$. By (A.19a) and (A.20a), it follows that:

$$\begin{aligned} w(\tau) &= w(\tau_0) + w(\sigma_{k-1} : n_{k-1} \rhd \sigma_{k-1} : n_{k-1} : \varepsilon(\mu(n)))\\ &+ w(\tau_1) + w(\sigma_{k-1} : n_{k-1} : n \rhd \sigma_{k-1} : n_k)\\ &= w(\tau_0) + 1 + 0 - 1\\ &= w(\pi[0..k-1]) + w(n_{k-1}\gamma_{k-1} \xrightarrow{\varphi_k} n_k\gamma_k) = w(\pi)\end{aligned}$$

By (A.19b) and (A.20b), it follows that:

$$\begin{array}{rcl}\Phi(\tau) & = & \Phi(\tau_0) \ \wedge \ \Phi(\sigma_{k-1} : n_{k-1} \rhd \sigma_{k-1} : n_{k-1} : \varepsilon(\mu(n))) \\ & \wedge & \Phi(\tau_1) \ \wedge \ \Phi(\sigma_{k-1} : n_{k-1} : n \rhd \sigma_{k-1} : n_k) \\ & = & \Phi(\pi[0..k-1]) \ \wedge \ \varphi_k \ = \ \Phi(\pi)\end{array}$$

We now prove that τ is positive. Let $i \in 0..l_0 + l_1 + 1$. We have to consider three cases. If $i \in 0..l_0 - 1$, then $w(\tau, 0, i) = w(\tau_0, 0, i) \geq 0$ follows directly from the fact that τ_0 is positive. Otherwise, if $i \in l_0..l_0 + l_1$, then, using also the fact that τ_1 is positive:

$$w(\tau, 0, i) \ = \ w(\tau_0) + w(\tau_0[k-1] \rhd \tau_1[0]) + w(\tau_1, 0, i - l_0) \ \geq \ 0$$

The last case ($i = l_0 + l_1 + 1$), is subsumed by the fact that $w(\tau) = w(\pi) \geq 0$.

Let $\sigma_k = \sigma_{k-1}$. We have still to prove that:

$$\flat(\tau) \ = \ (\sigma_0 : n_0)\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_k} (\sigma_k : n_k)\chi_k$$

Recall that τ is on the form:

$$\langle \sigma_0 : n_0, \chi_0 \rangle \rhd \cdots \rhd \langle \sigma_{l_0} : n_{l_0}, \chi_{l_0} \rangle \rhd \langle \sigma'_0 : n'_0, \chi'_0 \rangle \rhd \cdots \rhd \langle \sigma'_{l_1} : n'_{l_1}, \chi'_{l_1} \rangle \rhd \sigma_k : n_k$$

For $i \in 0..l_1$, let $\sigma_{l_0+i} = \sigma'_i$, $n_{l_0+i} = n'_i$ and $\chi_{l_0+i} = \chi'_i$. Let $l = l_0 + l_1 + 1$, $\sigma_l = \sigma_k$, $n_l = n_k$ and $\chi_l = \mathit{false}$. Therefore, τ can be rewritten as:

$$\langle \sigma_0 : n_0, \chi_0 \rangle \rhd \cdots \rhd \langle \sigma_l : n_l, \chi_l \rangle$$

By (A.20a), $w(\tau, l_0, l - 1) = 1 + w(\tau_1, 0, l_1) = 1$. Let $i \in l_0 + 1..l - 1$. Since τ_1 is positive, we have that:

$$w(\tau, i, l-1) \ = \ w(\tau, l_0, l-1) - w(\tau, l_0, i) \ = \ 1 - (1 + w(\tau_1, 0, i)) \ \leq \ 0$$

This proves that $l_0 = \max\{\, i \in 0..l-1 \mid w(\tau, i, l-1) = 1 \,\}$, i.e. $l_0 \rightleftarrows_\tau l - 1$. Since $w(\tau[l-1..l]) = -1$, the second rule in def. A.3 together with (A.19c) give:

$$\begin{array}{l}\flat(\tau) = \flat(\tau_0) \xrightarrow{\Phi(\tau[l_0..l])} \sigma_l : n_l \\ \qquad = (\sigma_0 : n_0)\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_{k-1}} \sigma_{k-1} : n_{k-1} \xrightarrow{\varphi_k} \sigma_l : n_l\end{array}$$

Since $n_{k-1} \dashrightarrow n_k$, constraint (1f) implies that it cannot be $n_{k-1} \longrightarrow n_k$, so $w(\pi[k-1] \rightarrow \pi[k]) = 0$. Then, (A.18b) is satisfied, because $\sigma_k = \sigma_{k-1}$. Moreover, by constraint (1d) it follows that:

$$\gamma_k \ = \ \gamma_{k-1} \ = \ \Gamma(\sigma_{k-1}) \cup \mathit{Dom}(n_{k-1}) \ = \ \Gamma(\sigma_k) \cup \mathit{Dom}(n_k)$$

- case $[pass]$. By the $\sharp_{pass}$ rule:

$$\frac{\ell(n_{k-1}) = \texttt{check}(P) \quad n_{k-1}\gamma_{k-1} \in N^\sharp \quad n_{k-1} \dashrightarrow n_k}{n_{k-1}\gamma_{k-1} \xrightarrow{\gamma_{k-1} \vdash P} n_k\gamma_{k-1}}$$

Then, $\gamma_k = \gamma_{k-1}$, $\varphi_k = \gamma_{k-1} \vdash P$ and $\chi_{k-1} = \chi_k = \mathit{false}$. By the induction hypothesis, there exists a positive trace τ'' such that $\langle G, \mathit{Perm}\rangle \triangleright \tau' \triangleright \tau''$, and:

$$\flat(\tau'') = (\sigma_0 : n_0)\chi_0 \xrightarrow{\varphi_1} \cdots \xrightarrow{\varphi_{k-1}} \sigma_{k-1} : n_{k-1}$$

Moreover, $w(\tau'') = w(\pi[0..k-1])$ and $\Phi(\tau'') = \Phi(\pi[0..k-1])$. Since, by assumption, $\mathit{Perm} \models \Phi(\pi) = \Phi(\pi[0..k-1]) \wedge (\gamma_{k-1} \vdash P)$, by definition 3.5 it follows that $\gamma_{k-1} \vdash_{\mathit{Perm}} P$. Then, equation (A.18c) and constraint (1e) imply that $\Gamma(\sigma_{k-1} : n_{k-1}) \vdash_{\mathit{Perm}} P$. So we can apply the $\triangleright_{pass}$ rule, that gives:

$$\frac{\ell(n_{k-1}) = \texttt{check}(P) \quad \Gamma(\sigma_{k-1} : n_{k-1}) \vdash_{\mathit{Perm}} P \quad n_{k-1} \dashrightarrow n_k}{\sigma_{k-1} : n_{k-1} \triangleright \sigma_{k-1} : n_k}$$

Let $\tau = \tau'' \triangleright \sigma_k = \sigma_{k-1} : n_k$. The induction hypothesis gives:

$$w(\pi) = w(\pi[0..k-1]) + 0 = w(\tau'') + 0 = w(\tau)$$
$$\Phi(\pi) = \Phi(\pi[0..k-1]) \wedge (\gamma_{k-1} \vdash P) = \Phi(\tau'') \wedge (\gamma_{k-1} \vdash P) = \Phi(\tau)$$

The proofs for (A.18a)–(A.18c) are trivial.

Proof of Theorem 3.7 Let $\pi = n_\varepsilon\emptyset \to \cdots \to n\gamma\chi$ be a path on $G^\sharp$, with $\mathit{Perm} \models \Phi(\pi)$. Let $\tau' = [\,]$, $\sigma_0 = [\,]$, $n_0 = n_\varepsilon$, $\chi_0 = \mathit{false}$ and $\gamma_0 = \emptyset$. Then, $\langle G, \mathit{Perm}\rangle \triangleright \tau' \triangleright \langle \sigma_0 : n_0, \chi_0\rangle$, and $\gamma_0 = \Gamma(\sigma_0) \cup \mathit{Dom}(n_0)$. By lemma A.9, there exists τ'' such that $\langle G, \mathit{Perm}\rangle \triangleright \tau' \triangleright \tau'' \triangleright \langle \sigma : n, \chi\rangle$ and $\Phi(\pi) = \Phi(\tau'' \triangleright \langle \sigma : n, \chi\rangle)$. Let $\tau = \tau' \triangleright \tau''$. Then, $\langle G, \mathit{Perm}\rangle \triangleright \tau \triangleright \langle \sigma : n, \chi\rangle$ and $\Phi(\pi) = \Phi(\tau \triangleright \langle \sigma : n, \chi\rangle)$.

Proof of Theorem 4.2 Consider first the "if" part. By contradiction, assume there exists a trace $\tau = [n_\varepsilon] \triangleright \cdots \triangleright \sigma : n$ on $\langle G, \mathit{Perm}\rangle$ such that $\Gamma(\sigma : n) \not\vdash P$. Then, by the $\triangleright_{fail}$ rule, $\langle G, \mathit{Perm}\rangle \triangleright \tau \triangleright \sigma : n\lightning$. Let $\gamma = \Gamma(\sigma : n)$. By constraint (1e), $\gamma = \Gamma(\sigma) \cup \mathit{Dom}(n)$. By theorem 3.6, there exists a path $\pi \in \Pi(n\gamma\lightning)$ in $G^\sharp$ such that $\Phi(\pi) = \Phi(\tau)$. By lemma A.8, $\mathit{Perm} \models \Phi(\pi)$. Since $\Phi(n\gamma\lightning) = \bigvee\{\, \Phi(\pi) \mid \pi \in \Pi(n\gamma\lightning) \,\}$, then $\mathit{Perm} \models \Phi(n\gamma\lightning)$ – contradiction.

For the "only if" part, let n be a redundant check for permission P, i.e. $\Gamma(\sigma : n) \vdash P$ whenever $\langle G, \mathit{Perm}\rangle \triangleright \sigma : n$ for some state σ. By contradiction, assume there exist a context γ such that $\mathit{Perm} \models \Phi(n\gamma\lightning)$, i.e. there exists a

path $\pi \in \Pi(n\gamma\lightning)$ such that $Perm \models \Phi(\pi)$. (definition 3.2). By theorem 3.7, there exist τ and σ such that $\langle G, Perm\rangle \rhd \tau \rhd \sigma : n\lightning$, $\Gamma(\sigma) \cup Dom(n) = \gamma$, and $\Phi(\tau \rhd \sigma : n\lightning) = \Phi(\pi)$. Moreover, the last step in the trace must be on the form:

$$\frac{\ell(n) = \texttt{check}(P) \quad \Gamma(\sigma : n) \not\vdash_{Perm} P}{\sigma : n \rhd \sigma : n\lightning}$$

because the only other rule leading to a state on the form $\sigma : n\lightning$ is $\rhd_{propagate}$, which however requires n being a call node (lemma A.2). Now, by definition 3.4:

$$\Phi(\tau \rhd \sigma : n\lightning) \;=\; \Phi(\tau) \wedge \big(\Gamma(\sigma : n) \not\vdash P\big) \;=\; \Phi(\tau) \wedge (\gamma \not\vdash P)$$

Since $Perm \models \Phi(\tau \rhd \sigma : n\lightning)$, by definition 3.5 it follows that $Perm \models \Phi(\tau)$ and $Perm \models \gamma \not\vdash P$. By constraint (1e), we have that $\gamma = \Gamma(\sigma) \cup Dom(n) = \Gamma(\sigma : n)$. Then, $\Gamma(\sigma : n) \not\vdash P$ – contradiction with the assumption of n being redundant.

Proof of Theorem 4.3 Let $n = \varepsilon(\mu(n))$ and $Perm \not\models \Phi(n\gamma)$ for all $n\gamma \in G^\sharp$. By contradiction, assume that $\langle G, Perm\rangle \rhd \sigma : m$ for some σ and $m \in \mu(n)$. Since $\mu(n_\varepsilon)$ has no entry points, it must be $\sigma \neq []$. So, σ is on the form $\sigma' : n'$. Consider the trace τ' such that $\langle G, Perm\rangle \rhd \tau' \rhd \sigma : m$. By lemma A.2, there exists an index i such that $\tau'[i] = \sigma' : n'$ and $n' \longrightarrow n$. Let $\tau = \tau'[0..i] \rhd \sigma : n$, and $\gamma = \Gamma(\sigma) \cup Dom(n)$. Then, lemma A.8 ensures that $Perm \models \Phi(\tau)$, and, by theorem 3.6, there exists a path $\pi \in \Pi(n\gamma)$ such that $\Phi(\pi) = \Phi(\tau)$. It follows that $Perm \models \Phi(n\gamma)$ – a contradiction.

Definition A.10 *The effect of inlining $\dot n \longrightarrow n'$ on context γ is defined as:*

$$Inl_{\dot n}(\gamma) = \begin{cases} \gamma & \text{if } Dom(n') \notin \gamma \\ (\gamma \setminus Dom(n')) \cup Dom(\dot n) & \text{otherwise} \end{cases}$$

Lemma A.11 *Let $\dot n$ be inlineable in G, and $\dot G = inl_{\dot n}(G)$. Then, for each state σ,*

$$\Gamma_{\dot G}(inl_{\dot n}(\sigma)) \;=\; Inl_{\dot n}(\Gamma_G(\sigma))$$

Proof. Let $\gamma = \Gamma_G(\sigma)$, $\dot\sigma = inl_{\dot n}(\sigma)$ and $\dot\gamma = \Gamma_{\dot G}(\dot\sigma)$. We proceed by induction on the size (number of nodes) of σ. The base case is $\sigma = []$. Then, $\dot\sigma = []$, $\gamma = \dot\gamma = \emptyset$, and $\emptyset = Inl_{\dot n}(\emptyset)$. For the inductive case, consider the last rule used in the derivation of $\dot\sigma = inl_{\dot n}(\sigma)$. We have the two following cases:

- if $\sigma = \sigma' : n'$ and $top(\sigma') \neq \dot n$, then $inl_{\dot n}(\sigma) = \dot\sigma' : n'$, where $\dot\sigma' = inl_{\dot n}(\sigma')$. Moreover, by condition (2b) of definition 4.4, it follows that $\dot n \not\longrightarrow \mu(n')$. Let $\gamma' = \Gamma_G(\sigma')$, $\dot\gamma' = \Gamma_{\dot G}(\dot\sigma')$. We have to consider the following two subcases.

If $Priv_G(n')$, then $\gamma = \{Dom_G(n')\}$. By definition 4.5, we have that $Priv_{\dot{G}}(n')$ and $Dom_{\dot{G}}(n') = Dom_G(n')$. Since $\dot{n} \not\longrightarrow \mu(n')$, definition A.10 implies that:

$$Inl_{\dot{n}}(\gamma) \;=\; Inl_{\dot{n}}(Dom_G(n')) \;=\; Dom_G(n') \;=\; Dom_{\dot{G}}(n') \;=\; \dot{\gamma}$$

Otherwise, if $\neg Priv_G(n')$, then:

$$\begin{array}{ll}
Inl_{\dot{n}}(\gamma) = Inl_{\dot{n}}(\gamma' \cup Dom_G(n')) & \text{by def. 2.3 } (\neg Priv_G(n')) \\
\quad = Inl_{\dot{n}}(\gamma') \cup Inl_{\dot{n}}(Dom_G(n')) & \text{by def. A.10} \\
\quad = \dot{\gamma}' \cup Inl_{\dot{n}}(Dom_G(n')) & \text{by the ind. hyp.} \\
\quad = \dot{\gamma}' \cup Inl_{\dot{n}}(Dom_{\dot{G}}(n')) & \text{by def. 4.5} \\
\quad = \dot{\gamma}' \cup Dom_{\dot{G}}(n') & \text{by def. A.10} \\
\quad = \dot{\gamma} & \text{by def. 2.3 } (\neg Priv_{\dot{G}}(n'))
\end{array}$$

- if $\sigma = \sigma' : \dot{n} : n'$, then $inl_{\dot{n}}(\sigma) = \dot{\sigma}' : n'$, where $\dot{\sigma}' = inl_{\dot{n}}(\sigma')$. Note that, by lemma A.2 and condition (2a) of definition 4.4, $\dot{n} \longrightarrow \mu(n')$. Let $\gamma' = \Gamma_G(\sigma')$ and $\dot{\gamma}' = \Gamma_{\dot{G}}(\dot{\sigma}')$. We have to consider the following two subcases.

  If $Priv_G(n')$, definition 4.5 states that $Priv_{\dot{G}}(n')$ and $Dom_{\dot{G}}(n') = Dom_G(\dot{n})$. Then, $\gamma = Dom_G(n')$ and $\dot{\gamma} = Dom_{\dot{G}}(n')$, so definition A.10 implies:

$$Inl_{\dot{n}}(\gamma) \;=\; Inl_{\dot{n}}(Dom_G(n')) \;=\; Dom_G(\dot{n}) \;=\; Dom_{\dot{G}}(n') \;=\; \dot{\gamma}$$

  Otherwise, if $\neg Priv_G(n')$, there are two further subcases, according $\dot{n}$ being privileged or not. If $Priv_G(\dot{n})$, then $Priv_{\dot{G}}(n')$ follows by definition 4.5, and:

$$\begin{array}{ll}
Inl_{\dot{n}}(\gamma) = Inl_{\dot{n}}(Dom_G(\dot{n}) \cup Dom_G(n')) & \text{as } Priv_G(\dot{n}),\ \neg Priv_G(n') \\
\quad = Dom_G(\dot{n}) & \text{by def. A.10} \\
\quad = Dom_{\dot{G}}(n') & \text{by def. 4.5} \\
\quad = \dot{\gamma} & \text{as } Priv_{\dot{G}}(n')
\end{array}$$

  Otherwise, if $\neg Priv_G(\dot{n})$, then:

$$\begin{array}{ll}
Inl_{\dot{n}}(\gamma) = Inl_{\dot{n}}(\gamma' \cup Dom_G(\dot{n}) \cup Dom_G(n')) & \text{as } \neg Priv_G(\dot{n}),\ \neg Priv_G(n') \\
\quad = Inl_{\dot{n}}(\gamma') \;\cup\; Dom_G(\dot{n}) & \text{by def. A.10} \\
\quad = \dot{\gamma}' \;\cup\; Dom_G(\dot{n}) & \text{by the ind. hyp.} \\
\quad = \dot{\gamma}' \;\cup\; Dom_{\dot{G}}(n') & \text{by def. 4.5} \\
\quad = \dot{\gamma} & \text{as } \neg Priv_{\dot{G}}(n')
\end{array}$$

Proof of Theorem 4.6 Let τ be on the form $\langle \sigma_0, \chi_0 \rangle \rhd \cdots \rhd \langle \sigma_k, \chi_k \rangle$, with $\sigma_0 = []$, $\chi_0 = \mathit{false}$. Then, $inl_{\dot{n}}(\tau)$ is on the form $\langle \dot{\sigma}_0, \chi_0 \rangle \rhd^{\dot{n}} \cdots \rhd^{\dot{n}} \langle \dot{\sigma}_k, \chi_k \rangle$,

where $\dot{\sigma}_i = inl_{\dot{n}}(\sigma_i)$ for each $i \in 0..k$. We have to prove that:

$$\langle \sigma_0, \chi_0 \rangle \ \rhd \ \cdots \ \rhd \ \langle \sigma_k, \chi_k \rangle \quad \Longleftrightarrow \quad \langle \dot{\sigma}_0, \chi_0 \rangle \ \rhd^{\dot{n}} \ \cdots \ \rhd^{\dot{n}} \ \langle \dot{\sigma}_k, \chi_k \rangle$$

Consider the forward implication first. We proceed by case analysis on the rule used to deduce $\sigma_i\chi_i \rhd \sigma_{i+1}\chi_{i+1}$. We omit a detailed discussion of the cases $\rhd_{fail}$ and $\rhd_{catch}$, because they are treated similarly to $\rhd_{pass}$ and $\rhd_{return}$, respectively.

- case $[call]$:

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n'}{\sigma : n \rhd \sigma : n : n'}$$

Here $\sigma_i = \sigma : n$, $\sigma_{i+1} = \sigma : n : n'$, and $\chi_i = \chi_{i+1} = false$. Let $\sigma' : n = inl_{\dot{n}}(\sigma : n) = \dot{\sigma}_i$. If $n \neq \dot{n}$, then rule $\rhd^{\dot{n}}_{call1}$ yields:

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n' \quad n \neq \dot{n}}{\sigma' : n \ \rhd^{\dot{n}} \ \sigma' : n : n'}$$

To show that $\dot{\sigma}_{i+i} = \sigma' : n : n' = inl_{\dot{n}}(\sigma : n : n') = inl_{\dot{n}}(\sigma_{i+1})$, it suffices to note that rule inl_2 instances to:

$$\frac{inl_{\dot{n}}(\sigma : n) = \sigma' : n \quad top(\sigma : n) \neq \dot{n}}{inl_{\dot{n}}(\sigma : n : n') = \sigma' : n : n'}$$

Otherwise, if $n = \dot{n}$, then rules $\rhd^{\dot{n}}_{call2}$ and inl_3 give:

$$\frac{\ell(\dot{n}) = \texttt{call} \quad \dot{n} \longrightarrow n'}{\sigma' : \dot{n} \ \rhd^{\dot{n}} \ \sigma' : n'} \qquad \frac{inl_{\dot{n}}(\sigma) = \dot{\sigma}}{inl_{\dot{n}}(\sigma : \dot{n} : n') = \dot{\sigma} : n'}$$

To prove that $\sigma' = \dot{\sigma}$, assume first that $\sigma = []$. Then, $\sigma' : \dot{n} = inl_{\dot{n}}([\dot{n}]) = [\dot{n}]$ implies that $\sigma' = []$, and $\dot{\sigma} = inl_{\dot{n}}([]) = []$,

Second, assume $\sigma = \sigma'' : n''$. Condition (2a) of definition 4.4 ensures that $n'' \neq \dot{n}$, because, otherwise, it would be $\dot{n} \longrightarrow \mu(\dot{n})$. Then, rule inl_2 gives:

$$\frac{inl_{\dot{n}}(\sigma'' : n'') = \dot{\sigma} \quad top(\sigma'' : n'') \neq \dot{n}}{inl_{\dot{n}}(\sigma'' : n'' : \dot{n}) = \dot{\sigma} : \dot{n}}$$

By assumption, it is also $\sigma' : \dot{n} = inl_{\dot{n}}(\sigma : \dot{n})$. Therefore, $\sigma' = \dot{\sigma}$.

- case $[return]$:

$$\frac{\ell(n') = \texttt{return} \quad n \dashrightarrow m}{\sigma : n : n' \rhd \sigma : m}$$

Let $\sigma' : n' = inl_{\dot{n}}(\sigma : n : n')$. We have to consider two subcases.

If $\dot{n} \not\longrightarrow \mu(n')$, let $\dot{\sigma} : n = inl_{\dot{n}}(\sigma : n)$. Then, lemma A.2 ensures that $n \neq \dot{n}$, hence rules inl_2 and $\rhd^{\dot{n}}_{return1}$ give:

$$\frac{inl_{\dot{n}}(\sigma : n) = \dot{\sigma} : n \quad top(\sigma : n) \neq \dot{n}}{inl_{\dot{n}}(\sigma : n : n') = \dot{\sigma} : n : n'} \qquad \frac{\ell(n') = \texttt{return} \quad n \dashrightarrow m \quad \dot{n} \not\longrightarrow \mu(n')}{\dot{\sigma} : n : n' \rhd^{\dot{n}} \dot{\sigma} : m}$$

Then, $\dot{\sigma} : m = inl_{\dot{n}}(\sigma : m)$ follows immediately by $\dot{\sigma} : n = inl_{\dot{n}}(\sigma : n)$.

Otherwise, if $\dot{n} \longrightarrow \mu(n')$, let $\dot{\sigma} = inl_{\dot{n}}(\sigma)$. Lemma A.2 and condition (2b) of definition 4.4 ensure that $n = \dot{n}$. Then, rules inl_3 and $\rhd^{\dot{n}}_{return2}$ give:

$$\frac{inl_{\dot{n}}(\sigma) = \dot{\sigma}}{inl_{\dot{n}}(\sigma : \dot{n} : n') = \dot{\sigma} : n'} \qquad \frac{\ell(n') = \texttt{return} \quad \dot{n} \dashrightarrow m \quad \dot{n} \longrightarrow \mu(n')}{\dot{\sigma} : n' \rhd^{\dot{n}} \dot{\sigma} : m}$$

To prove $\dot{\sigma} : m = inl_{\dot{n}}(\sigma : m)$, observe that, since $top(\sigma) \neq \dot{n}$ is ensured by condition (2a), then rule inl_2 instances to:

$$\frac{inl_{\dot{n}}(\sigma) = \dot{\sigma} \quad top(\sigma) \neq \dot{n}}{inl_{\dot{n}}(\sigma : m) = \dot{\sigma} : m}$$

- case $[pass]$:

$$\frac{\ell(n) = \texttt{check}(P) \quad \Gamma_G(\sigma : n) \vdash_{Perm} P \quad n \dashrightarrow m}{\sigma : n \rhd \sigma : m}$$

Let $\dot{\sigma} : n = inl_{\dot{n}}(\sigma : n)$ and $\dot{n} \longrightarrow n'$. By theorem 3.6 and constraint (1e), there exist $n\gamma \in G^\sharp$ such that $\gamma = \Gamma_G(\sigma : n)$. We must consider two cases. If $Dom(n') \notin \gamma$, then definition A.10 gives $Inl_{\dot{n}}(\gamma) = \gamma$. Therefore, by lemma A.11:

$$\Gamma_{\dot{G}}(inl_{\dot{n}}(\sigma : n)) \;=\; Inl_{\dot{n}}(\Gamma_G(\sigma : n)) \;=\; Inl_{\dot{n}}(\gamma) \;=\; \gamma$$

It follows that $\Gamma_{\dot{G}}(\dot{\sigma} : n) \vdash_{Perm} P$, so the transition is also possible in $\rhd^{\dot{n}}$.

Otherwise, if $Dom(n') \in \gamma$, then, by conditions (2a)–(2c) the state σ must be on the form $\sigma' : \dot{n} : n' : \sigma''$, for some σ', σ'' such that $\dot{n} \notin \sigma''$. By lemma A.2, it follows that $\langle G, Perm\rangle \rhd \sigma' : \dot{n}$. Consider the trace $\tau' = [] \rhd \cdots \sigma' : \dot{n}$, and let $\dot{\gamma} = \Gamma_G(\sigma') \cup Dom(\dot{n})$. Theorem 3.6 states that there exists a path $\pi' \in \Pi(\dot{n}\dot{\gamma})$ in $G^\sharp$. By $\sharp_{call}$, $\dot{n}\dot{\gamma} \rightarrow n'\gamma''$, where $\gamma'' = (\dot{\gamma} \uparrow \dot{n}) \cup Dom(n')$. Let h be the rightmost index of $\sigma' : \dot{n} : n'$ in τ, k be the rightmost index of $\sigma : n$, and $i \in h + 1..k$. It follows that σ_i is on the form $\sigma' : \dot{n} : n' : \sigma'_i$, for some $\sigma'_i \neq []$. Then, $w(\tau, h, i) = |\sigma'_i| > 0$, which, according to definition A.5,

means that the trace $\sigma' : \dot{n} : n' \rhd \cdots \rhd \sigma : n$ is positive. Therefore, we can apply lemma A.7 to find a path $\pi \in \Pi(n'\gamma'', n\gamma)$. This proves that $\dot{n}\dot{\gamma} \rightrightarrows n\gamma$. Since all the premises to condition (2d) in definition 4.4 are satisfied, we can conclude:

$$\begin{array}{lll}
\Gamma_G(\sigma : n) \vdash_{Perm} P & \iff \gamma' \cup Dom(n') \vdash_{Perm} P & \text{by def. } \gamma \\
& \iff \gamma' \vdash_{Perm} P \ \wedge \ P \in Perm(Dom(n')) & \text{by def. 2.4} \\
& \iff \gamma' \vdash_{Perm} P \ \wedge \ P \in Perm(Dom(\dot{n})) & \text{by (2d)} \\
& \iff \gamma' \cup Dom(\dot{n}) \vdash_{Perm} P & \text{by def. 2.4} \\
& \iff Inl_{\dot{n}}(\gamma) \vdash_{Perm} P & \text{by def. A.10} \\
& \iff \Gamma_{\dot{G}}(inl_{\dot{n}}(\sigma : n)) \vdash_{Perm} P & \text{by lemma A.11} \\
& \iff \Gamma_{\dot{G}}(\dot{\sigma} : n) \vdash_{Perm} P & \text{by def. } \dot{\sigma} : n
\end{array}$$

Therefore, $\Gamma_{\dot{G}}(\dot{\sigma} : n) \vdash_{Perm} P$, and the transition $\dot{\sigma} : n \rhd^{\dot{n}} \dot{\sigma} : m$ is possible. Note that $\dot{\sigma} : m = inl_{\dot{n}}(\sigma : m)$ immediately follows by $\dot{\sigma} : n = inl_{\dot{n}}(\sigma : n)$.

- case [*propagate*]:

$$\frac{n' \not\xrightarrow{}_{\lightning}}{\sigma : n'\lightning \rhd \sigma\lightning}$$

If $\dot{n} \not\longrightarrow \mu(n')$, let $\dot{\sigma} = inl_{\dot{n}}(\sigma)$. Lemma A.2 and condition (2a) ensure that $top(\sigma) \neq \dot{n}$. Then, rules inl_2 and $\rhd^{\dot{n}}_{propagate1}$ give:

$$\frac{inl_{\dot{n}}(\sigma) = \dot{\sigma} \quad top(\sigma) \neq \dot{n}}{inl_{\dot{n}}(\sigma : n') = \dot{\sigma} : n'} \qquad \frac{n' \not\xrightarrow{}_{\lightning} \quad \dot{n} \not\longrightarrow \mu(n')}{\dot{\sigma} : n'\lightning \ \rhd^{\dot{n}} \ \dot{\sigma}\lightning}$$

Otherwise, if $\dot{n} \longrightarrow \mu(n')$, then lemma A.2 and condition (2b) imply that $\sigma = \sigma' : \dot{n}$ for some σ'. Let $\dot{\sigma} = inl_{\dot{n}}(\sigma')$. Then, by rules inl_3 and $\rhd^{\dot{n}}_{propagate2}$:

$$\frac{inl_{\dot{n}}(\sigma') = \dot{\sigma}}{inl_{\dot{n}}(\sigma' : \dot{n} : n') = \dot{\sigma} : n'} \qquad \frac{n' \not\xrightarrow{}_{\lightning} \quad \dot{n} \longrightarrow \mu(n')}{\dot{\sigma} : n'\lightning \ \rhd^{\dot{n}} \ \dot{\sigma} : \dot{n}\lightning}$$

To prove that $\dot{\sigma} : \dot{n} = inl_{\dot{n}}(\sigma)$, observe that, since $top(\sigma') \neq \dot{n}$ is ensured by condition (2a), then rule inl_2 instances to:

$$\frac{inl_{\dot{n}}(\sigma') = \dot{\sigma} \quad top(\sigma') \neq \dot{n}}{inl_{\dot{n}}(\sigma' : \dot{n}) = \dot{\sigma} : \dot{n}}$$

For the backward implication, we proceed by case analysis on the rule used to deduce $\langle \dot{\sigma}_i, x_i \rangle \rhd \langle \dot{\sigma}_{i+1}, x_{i+1} \rangle$. The function inl is bijective: for each inlined state $\dot{\sigma}$, the original state can be recovered by inserting $\dot{n}$ before each

n' occurring in $\dot\sigma$ whenever $\dot n \longrightarrow \mu(n')$. A case analysis on the rule used for $\dot\sigma_i \rhd^{\dot n} \dot\sigma_{i+1}$ gives:

- case [*call1*]:

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n' \quad n \neq \dot n}{\sigma' : n \rhd^{\dot n} \sigma' : n : n'}$$

Since *inl* is bijective, let $\sigma : n$ be such that $inl_{\dot n}(\sigma : n) = \sigma' : n$. Then:

$$\frac{\ell(n) = \texttt{call} \quad n \longrightarrow n'}{\sigma : n \rhd \sigma : n : n'} \qquad \frac{inl_{\dot n}(\sigma : n) = \sigma' : n \quad top(\sigma : n) \neq \dot n}{inl_{\dot n}(\sigma : n : n') = \sigma' : n : n'}$$

follow by rules $\rhd_{call}$ and inl_2, respectively.

- case [*call2*]:

$$\frac{\ell(\dot n) = \texttt{call} \quad \dot n \longrightarrow n'}{\sigma' : \dot n \rhd^{\dot n} \sigma' : n'}$$

Let $\sigma : \dot n$ be such that $inl_{\dot n}(\sigma : \dot n) = \sigma' : \dot n$. Then:

$$\frac{\ell(\dot n) = \texttt{call} \quad \dot n \longrightarrow n'}{\sigma : \dot n \rhd \sigma : \dot n : n'} \qquad \frac{inl_{\dot n}(\sigma) = \sigma'}{inl_{\dot n}(\sigma : \dot n : n') = \sigma' : n'}$$

follow by rules $\rhd_{call}$ and inl_3, respectively.

- case [*return1*]:

$$\frac{\ell(n') = \texttt{return} \quad n \dashrightarrow m \quad \dot n \not\longrightarrow \mu(n')}{\dot\sigma : n : n' \rhd^{\dot n} \dot\sigma : m}$$

Since $\dot n \not\longrightarrow \mu(n')$, by condition (2a) it follows that $n \neq \dot n$. So, let $\sigma : n : n'$ be such that $inl_{\dot n}(\sigma : n : n') = \dot\sigma : n : n'$. Then:

$$\frac{\ell(n') = \texttt{return} \quad n \dashrightarrow m}{\sigma : n : n' \rhd \sigma : m} \qquad \frac{inl_{\dot n}(\sigma : n) = \dot\sigma : n \quad top(\sigma : n) \neq \dot n}{inl_{\dot n}(\sigma : n : n') = \dot\sigma : n : n'}$$

follow by rules $\rhd_{return}$ and inl_2, respectively, while $\dot\sigma : m = inl_{\dot n}(\sigma : m)$ immediately follows by the fact that $\dot\sigma : n = inl_{\dot n}(\sigma : n)$.

- case [*return2*]:

$$\frac{\ell(n') = \texttt{return} \quad \dot n \dashrightarrow m \quad \dot n \longrightarrow \mu(n')}{\dot\sigma : n' \rhd^{\dot n} \dot\sigma : m}$$

Let $\sigma : n'$ be such that $inl_{\dot{n}}(\sigma : n') = \dot{\sigma} : n'$. Since $\dot{n} \longrightarrow \mu(n')$, lemma A.2 and condition (2b) give that $top(\sigma) = \dot{n}$, i.e. $\sigma = \sigma' : \dot{n}$ for some σ'. Then:

$$\frac{\ell(n') = \texttt{return} \quad \dot{n} \dashrightarrow m}{\sigma' : \dot{n} : n' \rhd \sigma' : m} \qquad \frac{inl_{\dot{n}}(\sigma) = \dot{\sigma}}{inl_{\dot{n}}(\sigma : \dot{n} : n') = \dot{\sigma} : n'}$$

follow by rules $\rhd_{return}$ and inl_3, respectively, while $\dot{\sigma} : m = inl_{\dot{n}}(\sigma : m)$ immediately follows by the fact that $\dot{\sigma} = inl_{\dot{n}}(\sigma)$.

- case [*propagate1*]:

$$\frac{n' \not\dashrightarrow_{\lightning} \quad \dot{n} \not\longrightarrow \mu(n')}{\dot{\sigma} : n'\lightning \ \rhd^{\dot{n}} \ \dot{\sigma}\lightning}$$

Let $\sigma : n'$ be such that $inl_{\dot{n}}(\sigma : n') = \dot{\sigma} : n'$.

$$\frac{n' \not\dashrightarrow_{\lightning}}{\sigma : n'\lightning \rhd \sigma\lightning} \qquad \frac{inl_{\dot{n}}(\sigma) = \dot{\sigma} \quad top(\sigma) \neq \dot{n}}{inl_{\dot{n}}(\sigma : n') = \dot{\sigma} : n'}$$

follow by rules $\rhd_{propagate}$ and inl_2, respectively.

- case [*propagate2*]:

$$\frac{n' \not\dashrightarrow_{\lightning} \quad \dot{n} \longrightarrow \mu(n')}{\dot{\sigma} : n'\lightning \ \rhd^{\dot{n}} \ \dot{\sigma} : \dot{n}\lightning}$$

Let $\sigma : n'$ be such that $inl_{\dot{n}}(\sigma : n') = \dot{\sigma} : n'$. Since $\dot{n} \longrightarrow \mu(n')$, by lemma A.2 and condition (2b) there exists a σ' such that $\sigma = \sigma' : \dot{n}$. Then:

$$\frac{n' \not\dashrightarrow_{\lightning}}{\sigma : n'\lightning \rhd \sigma\lightning} \qquad \frac{inl_{\dot{n}}(\sigma') = \dot{\sigma}}{inl_{\dot{n}}(\sigma' : \dot{n} : n') = \dot{\sigma} : n'}$$

follow by rules $\rhd_{propagate}$ and inl_3, respectively, while $\dot{\sigma} : \dot{n} = inl_{\dot{n}}(\sigma)$ immediately follows by the fact that $inl_{\dot{n}}(\sigma') = \dot{\sigma}$.